TURING 图灵程序设计丛书

MANNING

The Well-Grounded Java Developer

Vital Techniques of Java 7 and Polyglot Programming

Java程序员修炼之道

[英] Benjamin J. Evans
[荷兰] Martijn Verburg 著
吴海星 译

涵盖Java 7最新特性

全面解读Groovy、Scala和Clojure在JVM上的应用

透视函数式编程的优势



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

TURING 图灵程序设计丛书



The Well-Grounded Java Developer

Vital Techniques of Java 7 and Polyglot Programming

Java程序员修炼之道

[英] Benjamin J. Evans
[荷兰] Martijn Verburg 著
吴海星 译

人民邮电出版社
北京

图书在版编目（CIP）数据

Java程序员修炼之道 / (英) 埃文斯 (Evans,B.J.),
(荷) 费尔堡 (Verburg,M.) 著 ; 吴海星译. -- 北京 :
人民邮电出版社, 2013.7
（图灵程序设计丛书）
书名原文: The well-grounded Java
developer:vital techniques of Java 7 and polyglot
programming
ISBN 978-7-115-32195-4

Ⅰ. ①J… Ⅱ. ①埃… ②费… ③吴… Ⅲ. ①
JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2013)第138623号

内 容 提 要

本书分为四部分，第一部分全面介绍 Java 7 的新特性，第二部分探讨 Java 关键编程知识和技术，第三部分讨论 JVM 上的新语言和多语言编程，第四部分将平台和多语言编程知识付诸实践。从介绍 Java 7 的新特性入手，本书涵盖了 Java 开发中最重要的技术，比如依赖注入、测试驱动的开发和持续集成，探索了 JVM 上的非 Java 语言，并详细讲解了多语言项目，特别是涉及 Groovy、Scala 和 Clojure 语言的项目。此外，书中含有大量代码示例，帮助读者从实践中理解 Java 语言和平台。

本书适合 Java 开发人员以及对 Java7 和 JVM 新语言感兴趣的各领域人士阅读。

◆ 著　[英] Benjamin J. Evans　[荷兰] Martijn Verburg
译　吴海星
责任编辑　刘美英
执行编辑　李　洁
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：26
字数：658千字　2013年 7 月第 1 版
印数：1－3 000册　2013年 7 月北京第 1 次印刷

著作权合同登记号　图字：01-2012-8794号

定价：89.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

序

“Kirk说加油站也卖啤酒。”这是Ben Evans跟我说的第一句话。他来克里特岛参加一个开放型Java会议。我说我通常到加油站就是加油，但那边拐角确实有个店卖啤酒。Ben看起来对我的回答有点儿失望。我在这个希腊小岛上生活了5年，还从来没在加油站买过啤酒。

当我在看这本书时，那种似曾相识的感觉又来了。我自认为是一名Java专家：用Java写了15年程序，发表了几百篇文章，在各种会议中演讲，还执教Java高级课程。可阅读Ben和Martijn的这本大作，经常能给我一些意料之外的启发。他们一开始先介绍了为改变Java生态系统所做的开发工作。类库的内部实现修改起来相对容易，一般也能见到成效。例如，`Arrays.sort()`的内部实现在Java 7中不再用MergeSort算法，而是改用了TimSort。由于这个变化，你不用修改自己对偏序数组进行排序的代码就可能看到性能的提升。然而，修改类文件格式或添加新的VM特性则需要大量工作。Ben了解这些情况，因为他在JCP执行委员会任职。这本书也是关于Java 7的，所以你能接触到Java 7中的所有新特性，比如语法糖的完善、`String`上的switch、分支/合并，还有Java NIO.2。

并发就是线程和同步，对吗？如果这就是你对多线程的认识，那么你需要学习新知识了。就像作者在书中指出的，“并发领域的研究工作正开展得热火朝天”。与并发相关的邮件列表上每天都有讨论，新点子层出不穷。本书会告诉你如何看待分而治之策略以及如何规避某些安全陷阱。

在我看到类加载那一章时，我觉得他们说得有点儿过了。那都是我和朋友们过去用来炫耀的技巧，居然也给摆出来供大家研习了！他们讲解了`javap`（这个小工具用于透视Java编译器生成的字节码）的工作方式，还谈到了新的`invokedynamic`指令，并解释了它跟普通反射的区别。

我特别喜欢讲性能调优的第6章。除了Jack Shirazi的*Java Performance Tuning*，这还是第一本能够抓住“如何使系统运行更快”这个本质问题的书。我可以用四个字来总结这一章的内容：“测量，别猜。”这是做好性能调优的本质，因为人们不可能猜到运行慢的是哪段代码。这一章从硬件的角度解读了性能方面的问题，而不是只提供编码技巧。作者还向你展示了如何测量性能。有一个挺有意思的基础测试小工具——`CacheTester`类，可用于查看缓存未命中时的开销。

本书第三部分介绍了JVM上的多语言编程。Java不仅仅是Java编程语言，它还是一个可以运行其他语言的平台。我们已经见过不同类型语言的爆炸式增长了。有些是函数式的，有些是声明式的，还有一些是平台的接口（Jython和JRuby），让其他语言可以在JVM上运行。语言分为动态的（如Groovy）和静态的（如Java和Scala）。在JVM上我们可能因为多种原因而使用非Java的语言。如果正好要开始一个新项目，在做决定之前先看看都有什么可用吧。你可能不用再写那么多套路化的代码了。

Ben和Martijn向我们介绍了三种备选语言：Groovy、Scala和Clojure。在我看来，它们是当下最切实可行的选择。作者描述了这些语言之间的差异、与Java的差异以及它们的特性。不需要太多的技术细节，介绍每种语言的各章足以帮你弄清楚应该用哪一种。别指望能在书中看到Groovy的参考手册，但你会了解哪种语言更适合你。

之后，你将深入了解如何进行测试驱动开发以及如何持续集成系统。我发现一件很有意思的事，忠实的“老管家”Hudson这么快就被Jenkins取代了。无论如何，这些工具跟Checkstyle和FindBugs一样都是项目管理的基本工具。

你有望通过研读本书成为一名优秀的Java开发人员。不仅如此，你还能了解如何保持优秀。Java一直在变。下一版中我们将见到lambda表达式和模块化。[①]人们也在不断设计新语言，不断更新并发结构。你现在了解的很多真相将来可能不再是真相。因此，我们必须活到老学到老！

一天，我又开车路过Ben想买啤酒的那个加油站。在经济状况如此低迷的希腊，它也像很多公司一样关张了。我再也不可能知道他们卖不卖啤酒了。

Heinz Kabutz博士

知名Java技术教育家、The Java Specialists’Newsletter创始人

① 实现Java模块化的Jigsaw项目被延后到Java 9了，至少要到2015年。——译者注

前　　言

本书最开始是给德意志银行外汇IT部的新人准备的培训笔记。Ben觉得市面上没有面向经验匮乏的Java开发人员的书，所以决定写一本来填补这个空白。

在德意志银行IT管理团队的支持下，Ben去了比利时的Devoxx会议寻找灵感。在那里他见到了IBM的三位工程师（Rob Nicholson、Zoe Slattery和Holly Cummins），他们把他引荐给了伦敦Java社区（LJC，伦敦Java用户组）。

接下来的周六正好是LJC组织的年度开放会议，就在那次会议上，Ben遇到了LJC的一位领导者——Martijn Verburg。两人一见如故，把酒言欢，惺惺相惜，大有相见恨晚之意。也正是两人对技术和教学的共同热爱促成了本书。

软件开发是一项社会活动，我们希望能借助本书唱响这一主题。我们认为，虽然在这项活动中技术占有很重要的地位，但人与人之间微妙的沟通和交互关系也不容忽视。要在书里轻松解释这些东西并不容易，但这一主题自始至终贯穿本书。

凭借着对技术的执着和对学习的热爱，开发人员孜孜不倦地工作着。我们希望本书讨论的一些话题能够激发他们的学习热情。这是一次观光之旅，而不是百科全书式的灌输，这就是我们的初衷：帮助你入门，然后让你自己去探索那些激发你想象力的东西。

本书不仅为大学毕业生准备了接引指南，更为所有心有困惑的Java开发人员提供了指导。因为他们都很想知道:“接下来我该学什么？未来要向什么方向发展？我要再好好考虑考虑!”

从Java 7的新特性到现代软件开发的最佳实践，再到平台的未来发展，本书一路向前，向你展示在成长为资深Java开发人员的过程中我们认为至关重要的那些知识。并发、性能、字节码和类加载是最让我们着迷的核心技术。我们还会谈到JVM上那些新的非Java语言（即多语言编程），因为在接下来的几年里，对于很多开发人员来说它们将变得越来越重要。

归根结底，这是一次以你和你的兴趣为核心的、具有前瞻性的旅程。我们认为成为一名优秀的Java开发人员有助于你彻底投入到工作中去并顺利驾驭开发，也有助于你对不断变化的Java世界及它的周边生态系统有更多了解。

我们希望这本“经验的结晶”对你来说既实用又有趣，希望它能让你深思，同时还能带给你快乐。无论如何，写这本书的体验确实如此!

致　谢

老话说得好，“众人拾柴火焰高”。对于本书来说，这句话非常贴切。如果没有朋友、亲人、同事、同行，甚至偶尔跟我们对立的那些人，我们就不可能完成本书。我们一直都非常幸运，因为那些对我们批评最强烈的人也可以算是我们的朋友。

帮助过我们的人太多了，很难全部列举出来。本书快付印的时候，我们在http://www.java7developer.com上发过一个帖子，其中也列出了一批名单，那些人值得我们感谢。

如果名单里遗漏了谁，都怪我们没有牢记您的大名，请接受我们的歉意！下面这些人对本书出版都有贡献，在此一并感谢（排名不分先后）。

伦敦 Java 社区

伦敦Java社区（LJC，www.meetup.com/londonjavacommunity）是我们两位作者相遇相知的地方。我们要感谢下面这些帮忙审校本书的人：Peter Budo、Nick Harkin、Jodev Devassy、Craig Silk、N. Vanderwildt、Adam J. Markham、“Rozallin”、Daniel Lemon、Frank Appiah、P. Franc、“Sebkom” Praveen、Dinuk Weerasinghe、Tim Murray Brown、Luis Murbina、Richard Doherty、Rashul Hussain、John Stevenson、Gemma Silvers、Kevin Wright、Amanda Waite、Joel Gluth、Richard Paul、Colin Vipurs、Antony Stubbs、Michael Joyce、Mark Hindess、Nuno、Jon Poulton、Adrian Smith、Ioannis Mavroukakis、Chris Reed、Martin Skurla、Sandro Mancuso和Arul Dhesiaseelan。

在Java语言之外，我们得到了James Cook、Alex Anderson、Leonard Axelsson、Colin Howe、Bruce Durling和Russel Winder博士非常认真的指导。在这里要特别感谢他们。

我们还要特别感谢LJC JCP委员会成员：Mike Barker、Trisha Gee、Jim Gough、Richard Warburton、Simon Maple、Somay Nakhal和David Illsley。

最后，感谢LJC的发起人Barry Cranford。四年前，他带领几个勇敢的人，怀抱一个梦想创建了LJC。现在，LJC已经有约2500名成员，并且很多其他技术社区也发源于它。LJC已经成为伦敦技术界的中流砥柱。

www.coderanch.com

我们要感谢Maneesh Godbole、Ulf Ditmer、David O’Meara、Devaka Cooray、Greg Charles、Deepak Balu、Fred Rosenberger、Jesper De Jong、Wouter Oet、David O’Meara、Mark Spritzler和

Roel De Nijs，因为他们提供了详细的评论和宝贵的反馈。

Manning出版社

感谢Manning的Marjan Bace接受我们这两个有着疯狂想法的作者。在制作本书出版的整个过程中，我们跟许多人打过交道，非常感谢他们：Renae Gregoire、Karen G. Miller、Andy Carroll、Elizabeth Martin、Mary Piergies、Dennis Dalinnik和Janet Vail。毫无疑问，还要感谢那些我们从未见过面的幕后工作者。没有他们，就没有这本书！

感谢Candace Gillhoolley在营销上的努力，还有Christina Rudloff和Maureen Spencer一直以来的支持。

感谢John Ryan III在本书付印前对书稿的全面技术审校。

感谢那些在本书编写的不同阶段阅读原稿并给编辑和我们提供宝贵反馈的审校者：Aziz Rahman、Bert Bates、Chad Davis、Cheryl Jerozal、Christopher Haupt、David Strong、Deepak Vohra、Federico Tomassetti、Franco Lombardo、Jeff Schmidt、Jeremy Anderson、John Griffin、Maciej Kreft、Patrick Steger、Paul Benedict、Rick Wagner、Robert Wenner、Rodney Bollinger、Santosh Shanbhag、Antti Koivisto和Stephen Harrison。

特别致谢

感谢Andy Burgess为本书开发的网站www.java7developer.com，非常棒。感谢实习生Dragos Dogaru测试了所有代码示例。

感谢Matt Raible非常友善地允许我们在第13章引用他关于如何选择Web框架的一些内容。

感谢Alan Bateman，他领导了Java 7的NIO.2。他的反馈对于向所有Java开发人员普及这个伟大“新API”至关重要。

Jeanne Boyarsky友好地充当了我们最优秀的技术把关者。她果然名不虚传，什么都躲不过她鹰一般犀利的眼睛。感谢Jeanne！

感谢Martin Ling非常细致地讲解了计时硬件，那是第4章介绍相关内容的主要原因。

感谢Jason Van Zyl，他非常友善地允许我们在第12章引用Sonatype Company出版的*Maven: The Complete Reference*中的一些内容。

感谢Kirk Pepperdine对第6章的反馈和意见，还有他的热情及对这个行业独到的见解。

感谢Heinz M. Kabutz博士为我们写的推荐序和他在克里特岛的盛情款待，还有非常赞的Java专家简报（www.javaspecialists.eu/）。

Ben Evans的致谢

许多人都以不同的方式为本书作出了贡献，但篇幅有限，就不一一致谢了。在此，特别感谢下面这些人。

感谢Bert Bates和其他Manning的员工，他们让我了解了稿件和书之间的区别。

感谢Martijn，当然，为了友谊，为了能帮我度过难关坚持写作，为了很多东西，总之一言难尽。

感谢我的家人，特别是我的外祖父John Hinton和祖父John Evans，我之所以成为我，是因为从他们那里继承了很多东西。

最后，感谢E-J（水獭在书中出现如此频繁的原因）和Liz，太多夜晚因为写作不能陪他们，感谢他们的理解。我的爱，献给他们。

Martijn Verburg 的致谢

感谢我的妈妈Janneke和爸爸Nico，感谢你们在我和姐姐小时候把Commodore 64电脑带回了家。尽管玩《跳跳人》(这真的是一个非常非常酷的平台游戏。老妈拿着操纵杆跑来跑去真是太好笑了:-))几乎成了我们在家里用电脑做得最多的事，但正是它的编程手册激发了我对技术的热情。爸爸还教会了我一个道理：如果你能把小事情做对，那么由小事情组成的大事也就不在话下了。我至今依然在编码和日常工作中将它奉为真理。

感谢我的姐姐Kim，感谢你小时候跟我一起写代码！我永远也忘不了第一个星域(缓慢地——那时，我可不太擅长做性能调优）出现在屏幕上时的场景，魔法真的显灵了！姐夫Jos给了我们俩很多灵感（不仅仅因为他是一位空间科学家，尽管那确实很酷！)。我那超级可爱的外甥女Gweneth也出现在了本书里，看你能不能找到她！

Ben是我认识的业内最棒的技术专家之一。有时他的技术水平可以用“吓人”来形容！我很荣幸能跟他合写本书，没想到关于JVM还有那么多知识我不知道。Ben一直都是LJC的优秀领导者，我俩在技术大会上一唱一和的演讲之后，甚至有人认为我们该一块去演喜剧了。能跟朋友合写一本书，真好。

最后，感谢我可爱的妻子Kerry，为了保证本书每一章的插图和屏幕截图都恰到好处，我们一次次地取消了活动计划，而你毫无怨言——你总是那么迷人。愿每个人都能拥有这样的爱和支持。

关 于 本 书

欢迎阅读本书。通过阅读本书，你将成为紧跟时代潮流的Java程序员，重燃对这一语言和平台的热情。学习过程中，你会发现Java 7的新特性，熟悉重要的现代软件技术（比如依赖注入、测试驱动开发和持续集成），并开始探索JVM上的非Java语言这个美丽新世界。

首先，我们来看看James Iry在他精彩的博文“A Brief, Incomplete, and Mostly Wrong History of Programming Languages”（简明、不完整并且漏洞百出的编程语言历史）中对Java语言的描述：

> 1996年，James Gosling发明了Java。Java相对繁琐、基于类，是支持垃圾收集、静态类型、单派发的面向对象语言，继承方式为实现单继承和接口多继承。Sun大肆宣扬Java的新颖性。

他对Java的描述基本上是在插科打诨，C#在文中也受到了同等待遇。但作为对一种语言的描述，这种方式也不赖。博文还有很多精彩之处，参见James的博客（http://james-iry.blogspot.com/）。没事的时候看看还是挺有收获的。

James的描述的确提出了一个很实际的问题。为什么我们还要讨论一种有将近16年历史的语言呢？它真的已经稳定，没有多少新东西或有意思的事情值得探讨了吗？

如果真是那样，本书就会很薄。事实是，我们依然在谈论Java，因为它的一大优点就是其在以下几个核心设计决策之上的构造能力，这些都已经在市场中获得了成功：

- 运行时环境的自动管理（比如垃圾收集、即时编译）；
- 语法简单，核心概念相对较少；
- 保守的语言进化方式；
- 在类库中增加功能和复杂性；
- 广泛、开放的生态系统。

这些设计决策一直在推动着Java世界的创新，简单的核心使得开发门槛很低，而广阔的生态系统使得后来者很容易找到适合自己需要的现成组件。

尽管从历史趋势上来看语言的变化很缓慢，但这些特质使得Java平台和语言既强大又充满活力。Java 7仍然延续了这一趋势。语言的改变是演进式，而不是革命式的。然而，Java 7跟之前版本相比有一个主要区别：它是第一个明确着眼于下一次发布的新版本。根据Oracle有关发布的“B计划”，Java 7为Java 8的主要变化打下了基础。

近年来，JVM上非Java语言的崛起也是一个重大变化。这引发了Java和其他JVM语言之间的相

互融合。现在有大量的项目完全运行在JVM之上（这个数量还在增加），而Java只是它们所用的编程语言之一。

多语言特别是涉及Groovy、Scala和Clojure语言的项目，是当前Java生态系统的一个重要因素，也是本书最后一部分的主题。

阅读须知

本书内容大体上适合顺序阅读，但我们也能理解某些读者想直奔主题的心情，因此也在一定程度上迎合了这种阅读需求。

我们非常认同自己动手的学习方法，所以建议读者在阅读的同时尝试示例代码。接下来介绍本书主要内容，希望习惯跳跃阅读的读者能从这里找到线索。

本书分四部分：

- 用Java 7做开发；
- 关键技术；
- JVM上的多语言编程；
- 多语种项目开发。

第一部分共两章，都是关于Java 7的内容。本书通篇使用Java 7的语法和语义，所以第1章“初识Java 7”是必读的。那些要处理文件、文件系统和网络I/O的开发人员应该会对第2章“新I/O”特别感兴趣。

第二部分共四章（第3~6章），涉及的主题包括依赖注入、现代并发、类文件/字节码以及性能调优。

第三部分共四章（第7~10章）介绍了JVM上的多语言编程。第7章是必读的，因为这一章介绍的JVM上可用语言的类型和使用是阅读后面章节的基础。接下来的三章分别介绍与Java类似的语言Groovy、兼具OO和函数式特色的混合语言Scala和纯函数式语言Clojure。刚接触函数式编程的开发人员可能需要按顺序阅读，但这几章本身是相互独立的，可以跳着读。

第四部分（最后四章）在之前内容基础上介绍了新内容。虽然各章可以独立阅读，但是在某些部分我们会假定你已经读过之前的内容，或者已经熟悉那些主题。

简言之，如果整本书你必看一章，那就看第1章。如果你会看第三部分，那一定要看第7章。其他各章既可以顺序阅读，也可以独立阅读，但后面的某些章节会假定你已经看过前面的内容。

读者对象

本书主要是为那些希望掌握Java语言和平台现代化知识的开发人员写的。如果你想跟上Java 7的步伐，就请阅读本书吧。

如果你想提升一下自己的技能，想搞清楚依赖注入、并发、测试驱动开发之类的主题，本书能为你打下良好的基础。

本书也是为已经认识到多语言编程趋势并想深入下去的开发人员准备的。具体来说，如果你想学习函数式编程，那么本书介绍Scala和Clojure的两章会很有帮助。

路线图

第一部分只有两章。第1章介绍了Java 7及其Coin项目，该项目包含很多小巧高效的特性。第2章全面介绍了新I/O API，包括对文件系统API的全面梳理，还介绍了新的异步I/O能力。

第二部分分四章介绍了Java 7的关键技术。第3章告诉你依赖注入技术的源流，接着展示了Java中的标准解决方案Guice 3。第4章阐述在Java中如何正确进行现代并发开发。因为硬件行业坚定地朝着多核处理器方向发展，这个话题再次成为焦点。第5章介绍了JVM的类文件和字节码，揭示了它们的秘密，让你明白Java的工作原理。第6章讲解Java应用程序调优的基础知识，并讨论垃圾收集器等内容。

第三部分介绍JVM上的多语言编程，由四章组成。多语言编程的内容从第7章开始，这里讲述了多语言编程背景知识，以及使用另一种语言的恰当时机。第8章介绍了Groovy——Java动态编程的朋友。Groovy突显了语法相似的动态语言如何大幅提升Java开发人员的生产率。第9章将你带入函数式/OO混合的Scala世界。Scala是一种强大精炼的语言。第10章是为Lisp粉丝们准备的。Clojure被广泛誉为“使用得当的Lisp”，它全面展示了JVM上函数式语言的力量。

第四部分以前三部分的内容为基础，讨论多语言编程技术在几个编程领域涉及的问题。第11章谈到了测试驱动开发，还提供了一个围绕处理模拟对象的方法，给出了一些实战建议。第12章介绍了两种得到广泛应用的工具，用于构建流程中的Maven 3和用于持续集成的Jenkins/Hudson。第13章涵盖了与快速Web开发相关的主题，解释了Java在这一领域的传统缺陷，并提供了一些原型化的新技术（Grails和Compojure）。第14章是对全书的总结和对未来的展望，其中包括Java 8可能支持的新功能。

代码约定及下载

首先需要下载和安装的是Java 7。只要找到适合你的OS的发布包，按照下载和安装说明来做就行了。Oracle网站上Java SE部分有安装包的在线下载和说明，网址是www.oracle.com/technetwork/ java/javase/downloads/index.html。

另请参见附录A中的安装说明以及源码运行的指南。

书中所有源码都是等宽字体，以区别于周围的文字。很多代码清单中都有注解，指出其中的关键概念，有时候文中使用带圆圈的数字，对代码进行更详细的注解。我们尽量按照版心的宽度设置代码格式，也在必要时换行，并谨慎使用缩进。但有时候代码行太长，因此会有续行标记。

书中所有示例的源码都可在www.manning.com/TheWell-GroundedJavaDeveloper找到。[①]书中

① 本书源码也可在图灵社区http://www.ituring.com.cn/book/1027下载。——编者注

自始至终都有示例代码。比较长的代码清单有清楚的清单标题，短一些的代码清单就在文字的行与行之间。

软件需求

如今，Java 7几乎可运行在任何现代平台上。只要你使用以下某个操作系统，就可以运行源码示例：

- MS Windows XP及以上版本；
- 较新版的*nix；
- Mac OS X 10.6及以上版本。

多数人都想在IDE中试验代码示例。以下这些主流IDE都对Java 7和最新版的Groovy、Scala、Clojure提供良好支持：

- Eclipse 3.7.1及以上版本；
- NetBeans 7.0.1及以上版本；
- IntelliJ 10.5.2 及以上版本。

我们使用了NetBeans 7.1和Eclipse 3.7.1来创建和运行代码示例。

作者在线

购买本书英文版的读者可以免费访问由Manning出版社运营的专用Web论坛，并在论坛中对该书进行评论、提出技术问题、从作者和其他用户那里得到帮助。要访问并订阅该论坛，请访问www.manning.com/TheWell-GroundedJavaDeveloper，这个页面介绍了注册后如何访问论坛、可以得到什么帮助以及论坛上的行为规则。

Manning向读者承诺为读者之间、读者与作者之间提供一个可以对话的场所，但不会强制作者参与，作者在论坛上的贡献都是自愿而且不收费的。为使作者感兴趣，提高其参与度，我们建议读者向作者提一些具有挑战性的问题。

只要本书英文版仍然在售，读者就可以从出版社的网站上访问作者在线论坛和之前讨论话题的归档。

关于作者

Ben Evans是伦敦Java用户组的发起人、协助定义Java生态系统标准的Java社区过程执行委员会成员。他在技术圈已经度过了很多年"有趣的时光"，现为一家面向金融业的Java技术公司的CEO。Ben经常在公开场合发表关于Java平台、性能和并发的演讲。

Martijn Verburg（是jClarity的CTO）作为一名技术专家和众多初创企业的OSS导师，拥有十多年的经验。他也是伦敦Java用户组的领导者，带领全球的Java用户组成员为JSR（采用JSR计划）和OpenJDK（采用OpenJDK计划）作出了贡献。作为一位公认的技术团队优化专家，他经常应邀出席Java界的大型会议（JavaOne、Devoxx、OSCON、FOSDEM等）并发表演讲，人送雅号"开发魔头"，赞颂他敢于向行业现状挑战的精神。

关于封面图片

本书封面上的画像标题为“卖花人”，摘自19世纪法国出版的沙利文·马雷夏尔（Sylvain Maréchal）四卷本的地域服饰风俗纲要。其中每幅插图都是手工精心绘制并上色的。马雷夏尔这套书展示的丰富服饰，令我们强烈感受到200年前乡村与城镇的巨大文化差异。不同地域的人山水阻隔，言语不通。无论奔走于街巷，还是驻足于乡间，通过他们的服饰，一眼就能看出他们的生活场所、职业，以及生活境况。

时过境迁，书中描绘的那些区域性服饰差异到如今已经不复存在。即使是不同国家，都很难再看出人们着装的区别，再不必说城镇和乡村了。或许，我们今天多姿多彩的人生，正是从前那些文化差异的体现。只不过，如今的生活更加多元，而且技术环境下的生活节奏也更快了。

今时今日，计算机图书层出不穷，Manning就以马雷夏尔这套书中多样性的图片，来表达对IT行业日新月异的发明与创造的赞美。

目　　录

第三部分　JVM 上的多语言编程

第四部分　多语种项目开发

Part 1 第一部分

用 Java 7 做开发

本书前两章主要讨论 Java 7 的高明之处。为便于读者理解下文，第 1 章先介绍了一些可提高开发人员工作效率的语法变化，这些变化并不大，但效果都比较显著。第 1 章在这一部分中主要起抛砖引玉的作用，而另一个主题，Java 中的新 I/O 才是主角。

优秀的 Java 开发人员要了解语言的新特性。Java 7 中的新特性可以使开发人员的工作变得更轻松。但对于这些新变化，光了解语法是不够的。为了能迅速写出高效、安全的代码，你还需要对实现这些新特性的原因和方式有深刻的认识。Java 7 的变化可以大致分为两块：Coin 项目和 NIO.2。

第一块是 Coin 项目，包括语言层面的一些小变化，设计它们的初衷是提高开发人员的生产率，但又不会对底层平台造成太大影响。这些变化包括：

- try-with-resources 结构（可以自动关闭资源）；
- `switch` 中的字符串；
- 对数字常量的改进；
- `Multi-catch`（在一个 `catch` 块中声明多个要捕获的异常）；
- 钻石语法（在处理泛型时不用那么繁琐了）。

这些变化看起来都不大，但探索这些简单的语法修改背后的语义迁移，能让你洞察 Java 语言和 Java 平台之间的差别。

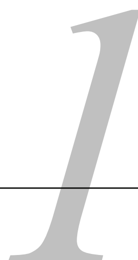
第二块变化是新 I/O（NIO.2）API，跟 Java 原有的文件系统支持相比，它具有压倒性优势，还提供了强大的异步能力。这些变化包括：

- 用于引用文件和类文件实体的新 `Path` 结构；
- 简化文件的创建、复制、移动和删除的工具类 `Files`；
- 内建的目录树导航；
- 在后台处理大型 I/O 的将来式和回调式异步 I/O。

第一部分结束时，你会很自然地用 Java 7 的方式来思考问题和编写代码。我们在后续章节中还会用到 Java 7 中的新特性，所以你还有机会不断温习这些新知识。

第 1 章

初识Java 7



本章内容

- ❑ Java既是编程语言，也是平台
- ❑ 语法变一点，能力强好多
- ❑ try-with-resources语句
- ❑ 提升异常处理能力

欢迎进入Java 7的世界！斗转星移，时过境迁。当尘埃落定，我们终于见到了Java 7的真容。虽然看起来有点陌生，但它必将带来全新的体验！跟随我们经历过这段探索之旅，你将进入更广阔的世界，发现更多新特性、更高明的编程技巧，并接触到JVM上运行的更多编程语言。

现在，我们先来热热身。虽然只是简单介绍，但还是能让你了解Java 7的强大特性。我们会先解释Java语言和平台的区别，因为有时人们会对这两种说法产生误解。

接着我们会介绍Coin项目，它汇聚了Java 7里短小精悍的新特性。我们会向你展示Java平台所接受、吸纳和发布的那些特性，就是它们构成了Java的变化。在此之后，我们会介绍Coin项目中新引入的6个主要特性。

你会学到新的语法，比如改进的异常处理方式（multi-catch）以及try-with-resources结构，借此在处理文件或其他资源的代码中躲开那些烦人的bug。读完本章，你将能用全新的方式编写Java代码，并整装待发，准备好迎接更大的挑战!

让我们先来探讨一下Java作为语言和平台的双重角色，这是现代Java的核心。这个知识点将贯穿全书，是一个必须掌握的基本要点。

1.1　语言与平台

使用Java之前，我们要先弄清楚Java语言和Java平台之间的区别。然而，有时候不同的作者对语言和平台的构成会有不同的定义，所以人们有时不太清楚两者之间的区别，分不清是语言还是平台提供了代码使用的编程特性。

因为本书的大部分内容都需要你理解两者的区别，所以这里需要说明一下。以下是我们给出的定义。

- Java语言　在“关于本书”中，我们提到Java语言是静态类型、面向对象的语言，希望你对这种说法已经非常熟悉了。Java语言还有一个非常明显的特点，它是（或者说应该是）人类可读的。
- Java平台　平台是提供运行时环境的软件。Java虚拟机（JVM）负责把类文件形式（人类不可读）的代码链接起来并执行。JVM不能直接解释Java语言的源文件，你要先把源文件转换成类文件。

Java作为软件系统之所以能成功，主要因为它是一种标准。也就是说，它有规范文件描述它应该如何工作。不同的厂商或项目组可以据此推出自己的实现，这些不同实现的工作方式在理论上是相同的。规范虽然不能保证这些实现处理同一任务时表现如何，但可以保证处理结果的正确性。

控制Java系统的规范有多种，其中最重要的是《Java语言规范》（JLS）和《JVM规范》（VMSpec）。在Java 7中，这两者之间的界限愈发清晰。实际上，VMSpec不再引用JLS中的任何内容，如果你认为这是Java 7重视Java之外其他语言的信号，说明你有见微知著的能力！希望你能继续关注，接下来我们会更加深入地探讨这两个规范之间的差别。

提到Java的双重角色，你自然想问：“它们两者之间还有什么关联吗？”如果它们在Java 7中如此泾渭分明，又是如何共同形成我们所熟悉的Java系统的呢？

连接Java语言和平台之间的纽带是统一的类文件（即.class文件）格式定义。认真研究类文件的定义能让你获益匪浅，这是优秀Java程序员向伟大Java程序员转变的一个途径。图1-1展示了产生和使用Java代码的整个过程。

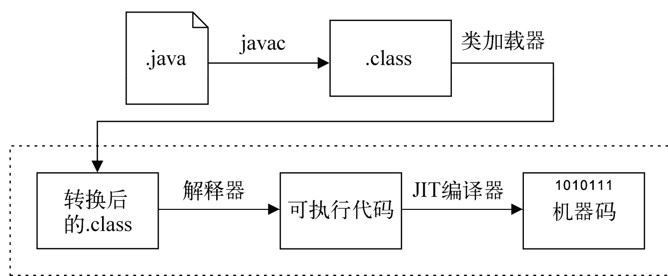


图1-1　Java源码被转换成 .class 文件，在JIT编译前被加载处理

如图所示，Java代码的演进过程从我们可以看懂的Java源码开始，然后由`javac`编译成.class文件，变成可以加载到JVM中的形式。值得注意的是，类文件在加载过程中通常都会被处理和修改。大多数流行框架（特别是打着“企业级”旗号的）都会在类加载过程中对类进行改造。

Java是编译型语言还是解释型语言？

大多数开发人员都知道，Java源文件需要编译成.class文件才能在JVM中运行。如果继续追问，许多开发人员还会告诉你说.class中的字节码首先会被JVM解释，但在稍后即时（JIT）编译。然而很多人将字节码含糊地理解为“在某种虚构的或简化的CPU上运行的机器码”。

实际上，JVM字节码更像是中途的驿站，是一种从人类可读的源码向机器码过渡的中间状态。用编译原理术语讲，字节码实际上是一种中间语言（IL）形态，不是真正的机器码。也就是说，将Java源码变成字节码的过程不是C或C++程序员所理解的那种编译。Java所谓的编译器javac也不同于gcc，实际上它只是一个针对java源码生成类文件的工具。Java体系中真正的编译器是JIT，如图1-1所示。

有人说Java是“动态编译”的，他们所说的编译是指JIT的运行时编译，不是指构建时创建类文件的过程。

所以如果被问及“Java是编译型语言还是解释型语言”，你可以回答“都是”。

希望我们已经把Java语言和Java平台之间的区别解释清楚了。接下来我们进入下一话题，看看Java 7中一些语法上的调整，先从Coin项目中的那些小变化开始。

1.2 Coin 项目：浓缩的都是精华

自2009年1月起，Coin便是Java 7（和Java 8）中一个开源的子项目。本节，我们会以Coin项目中包含的小变化为例，解释一下Java语言如何演进以及那些特性是如何被选中的。

为Coin项目命名

创建Coin项目是为了反映Java语言中的微小变动。项目的名字是个双关语——像硬币一样小的变化（small change comes as coins），而“套用一句老话”（to coin a phrase）指的是给Java语言添一个新的表述方式。

在技术圈子里，这种文字游戏、奇思妙想和躲不掉的恐怖双关语随处可见。你可能已经对此习以为常了。

我们觉得解释语言“为什么要变”和“变成了什么”同样重要。在开发Java 7的过程中，人们对新语言特性产生了很多兴趣，但Java社区有时并不明白要按时实现这些特性需要多大工作量。希望我们在“为什么要变”这一问题上能够对你有所启示，也希望能借此消除一些荒诞的说法。如果你对Java如何发展进化不感兴趣，可以直接阅读1.3节，看看Java语言发生了哪些变化。

关于修改Java语言有一个工作量曲线，某些实现方式可能要比其他方式需要的工作量少。如图1-2所示，我们尽量把不同的修改方式及其所需的工作量呈现出来，以展现不同复杂度及相关工作量的增长。

一般来说，工作量越少越好。也就是说，如果能用类库实现新特性，那就应该用类库。但有些特性用类库或增加IDE功能实现起来有难度，甚至根本不可能，那就必须在平台的更深层中实现。

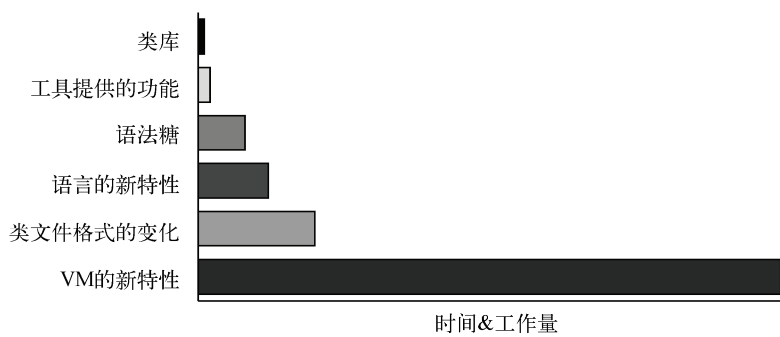


图1-2 用不同方式提供新功能所需工作量的比较

下面是一些特性（大部分是Java 7中的），它们符合我们为语言新特性定义的复杂度。

- 语法糖——数字中的下划线（Java 7）；
- 新的语言小特性——try-with-resources（Java 7）；
- 类文件格式的变化——注解（Java 5）；
- JVM的新特性——动态调用（Java 7）。

语 法 糖

“语法糖”是描述一种语言特性的短语。它表示这是冗余的语法——在语言中已经存在一种表示形式了——但语法糖用起来更便捷。

一般来说，程序的语法糖在编译处理早期会从编译结果中移除，变为相同特性的基础表示形式，这称为“去糖化”。

因此，语法糖是比较容易实现的修改。它们通常不需要做太多工作，只需要修改编译器（对于Java来说就是javac）。

Coin项目中（以及本章余下的内容）都是关于从Java语法糖到小的新特性这个范围之内的变化。

Coin项目的最初建议阶段从2009年2月持续到3月，coin-dev的邮件列表上收到了大约70个提议，囊括了各种可能的改进。甚至有人开玩笑说要增加lolcat风格的多行字符串（把标题叠加在好笑或生气的猫咪图片上，看你怎么想了——http://icanhascheezburger.com/）。

Coin项目提案的评判规则很简单。贡献者要完成三项任务：

- 提交一份详细的提案来描述修改（本质上应该是对Java语言的修改，而不是针对虚拟机的）；
- 在邮件列表上针对提案进行开放式讨论，能够接受其他参与者建设性的批评和建议；
- 随时可以提供一组能够实现变化的补丁原型。

Coin项目很好地示范了语言和平台在未来如何演进，所有的修改都会公开讨论，提供特性的早期原型，并且呼吁公众参与。

“什么是对规范的小修改？”现在也许就是提出这个问题的最好时机。我们马上要讨论在JLS的第14.11节中增加的一个单词——`"String"`。你可能再也找不出比这个更小的修改了，可即便是这样一个修改都会涉及规范中其他几个地方。

Java 7是以开源方式开发后发布的第一个版本

Java一开始并不是开源语言，但在2006年的JavaOne大会上其自身的源码以GPLv2许可发布（去掉了一些Sun不具有版权的源码）。当时正值Java 6发布前后，所以Java 7是Java在开源软件（OSS）许可下发布的第一版。开源的Java平台开发主要集中在项目OpenJDK上。

邮件列表coin-dev、lambda-dev和mlvm-dev等是讨论未来各种可能特性的主要场所，来自五湖四海的开发人员都可以借此参与到创造Java 7的过程中来。实际上，我们参与领导了“Adopt OpenJDK”计划，为新加入OpenJDK的开发人员提供指导，帮助改进Java。如果你想加入我们，请访问http://java.net/projects/jugs/pages/AdoptOpenJDK。

任何修改都会产生影响，我们必须从Java语言的整体设计上来追踪这些影响。

任何修改都应该严格执行下面这些操作（或至少调研一下）：

- 更新JLS；
- 在源码编译器中实现一个原型；
- 为修改增加必要的类库支持；
- 编写测试和示例；
- 更新文档。

除此之外，如果修改触及VM或者平台，应该：

- 更新VMSpec；
- 实现VM的修改；
- 在类文件和VM工具中增加支持；
- 考虑对反射的影响；
- 考虑对序列化的影响；
- 想一想对本地代码组件的影响，比如Java本地接口（JNI）。

考虑到这些修改对整体语言规范产生的影响，这可不是一星半点儿的工作。

如果你要修改类型系统，简直就是自寻死路，类型系统可是个不折不扣的雷区。这不是因为Java的类型系统不好，而是因为对于拥有多种静态类型系统的语言来说，这些类型系统之间有可能会产生很多交叉点。修改它们很容易出现意想不到的状况。

Coin项目选的路线非常明智，它建议贡献者们在修改提案中远离类型系统。因为对这种修改而言，即使最小的变化都需要做大量的工作，所以这种做法也是比较务实的。

在简单介绍了Coin项目的背景之后，接下来该看看它包含哪些特性了。

1.3 Coin 项目中的修改

Coin项目主要给Java 7引入了6个新特性，它们分别是switch语句中的String、数字常量的新形式、改进的异常处理、try-with-resources、钻石语法，还有变参警告位置的修改。

我们会详细讲解Coin项目中的这些变化，讨论这些新特性的语法和含义，并尽可能解释推出这些特性背后的动机。当然，我们也不是要把提案全部照搬过来，coin-dev邮件列表的归档里有完整的提案，如果你是一个好奇的语言设计师，可以去那里看看，还可以和大家讨论你的想法。

闲言少叙，开始介绍第一个Java 7新特性——switch语句中的String值。

1.3.1 **switch语句中的String**

switch语句是一种高效的多路语句，可以省掉很多繁杂的嵌套if判断，比如像这样：

```
public void printDay(int dayOfWeek) {
  switch (dayOfWeek) {
    case 0: System.out.println("Sunday"); break;
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
    case 4: System.out.println("Thursday"); break;
    case 5: System.out.println("Friday"); break;
    case 6: System.out.println("Saturday"); break;
    default: System.err.println("Error!"); break;
  }
}
```

在Java 6及之前，case语句中的常量只能是byte、char、short和int（也可以是对应的封装类型 Byte、Character、Short和Integer）或枚举常量。Java 7规范中增加了String，毕竟它也是常量类型。

```
public void printDay(String dayOfWeek) {
  switch (dayOfWeek) {
    case "Sunday": System.out.println("Dimanche"); break;
    case "Monday": System.out.println("Lundi"); break;
    case "Tuesday": System.out.println("Mardi"); break;
    case "Wednesday": System.out.println("Mercredi"); break;
    case "Thursday": System.out.println("Jeudi"); break;
    case "Friday": System.out.println("Vendredi"); break;
    case "Saturday": System.out.println("Samedi"); break;
    default: System.out.println("Error: '"+ dayOfWeek
    ➥ +"' is not a day of the week"); break;
  }
}
```

除此之外，switch语句和以前完全一样。像Coin项目中的许多新特性一样，这不过是一个让你更轻松的小小改进。

1.3.2　更强的数值文本表示法

当时有几个与整型语法相关的提案，最终被选中的是下面这两个：

- 数字常量（如基本类型中的integer）可以用二进制文本表示；
- 在整型常量中可以使用下划线来提高可读性。

这两个改变乍看起来都不起眼，但它们确实解决了一直困扰着Java程序员的一些小麻烦。

这两个新特性对系统底层程序员，就是那些整天处理原始网络协议、加密或沉迷于摆弄比特的人们特别有用。先来看一下二进制文本。

1. 二进制文本

在Java 7之前，如果要处理二进制值，就必须借助棘手（又容易出错）的基础转换，或者调用parseX方法。比如说，如果想让int x用位模式表示十进制值102，你可以这样写：

```
int x = Integer.parseInt("1100110", 2);
```

为了确保x是正确的位模式，你需要敲许多代码。这种方式尽管看起来还行，但实际上存在很多问题：

- 十分繁琐；
- 方法调用对性能有影响；
- 需要知道parseInt()的双参形式；
- 需要记住双参的parseInt()的处理细节；
- JIT编译器更难实现；
- 用运行时的表达式表示应该在编译时确定的常量，导致x不能用在switch语句中；
- 如果在位模式中有拼写错误（能通过编译），会在运行时抛出RuntimeException 。

现在好了，用Java 7可以写成：

```
int x = 0b1100110;
```

我们没说这种方法无所不能，但它确实解决了上面提到的那些问题。

你在跟二进制打交道时，这个小特性会是你的得力助手。比如在处理字节时，可以在switch语句中使用由位模式定义的二进制常量。

另外一个新特性虽然小，但却很实用——可以在表示一组二进制位或其他长数值的数字中加入下划线。

2. 数字中的下划线

众所周知，人脑和电脑有很多不同的地方，对于数字的处理方式就是其中之一。通常人们都不太喜欢面对一大串数字。这也是我们发明十进制的原因之一——因为人脑更擅于处理信息量大的短字串，而不是每个字符信息量都不太多的长字串。

也就是说，我们觉得1c372ba3要比00011100001101110010101110100011更容易处理，但电脑只认第二种。人们在处理长串数字时会采用分隔法，比如用404-555-0122表示电话号码。

注意 如果你跟作者（欧洲人）一样，想知道为什么美国电影或书里的电话号码总是以555开头，我可以告诉你。555-01xx是保留号段，用于虚构的情境。这是为了避免现实生活中的人接到那些对好莱坞电影过分投入的人打来的电话。

其他带有分隔符的一长串数字：

- 100 000 000美元（一大笔钱）；
- 08-92-96（英国银行的排序代码）。

可在代码中处理数字时不能用逗号（,）和连字符（-）作分隔符，因为它们可能会引发歧义。Coin项目中的提案借用了Ruby的创意，用下划线（_）作分隔符。注意，这只是为了让你阅读数字时更容易理解而做的一个小修改，编译器会在编译时把这些下划线去掉，只保留原始数字。

也就是说，为了不把100 000 000 和10 000 000搞混，你可以在代码中将100 000 000写成100_000_000，以便很容易区分它和10_000_000的差别。来看下面两个例子，至少你应该对其中一个比较熟悉：

```
long anotherLong = 2_147_483_648L;
int bitPattern = 0b0001_1100__0011_0111__0010_1011__1010_0011;
```

注意：赋给anotherLong的数值现在看起来清楚多了。

警告 在Java中可以用小写字母l表示长整型数值，比如1010100l。但最好还是用大写字母L，以免维护代码的人把数字1和字母l搞混：1010100L看起来要清楚得多。

现在你应该清楚这些变化给整数处理带来的好处了！让我们继续前进，去看看Java 7中的异常处理。

1.3.3 改善后的异常处理

异常处理有两处改进——multicatch和final重抛。要知道它们对我们有什么帮助，请先看一段Java 6代码。下面这段代码试图查找、打开、分析配置文件并处理此过程中可能出现的各种异常：

代码清单1-1 在Java 6中处理不同的异常

```
public Configuration getConfig(String fileName) {
  Configuration cfg = null;
  try {
    String fileText = getFile(fileName);
    cfg = verifyConfig(parseConfig(fileText));
  } catch (FileNotFoundException fnfx) {
    System.err.println("Config file '" + fileName + "' is missing");
  } catch (IOException e) {
    System.err.println("Error while processing file '" + fileName + "'");
  } catch (ConfigurationException e) {
    System.err.println("Config file '" + fileName + "' is not consistent");
```

```
    } catch (ParseException e) {
      System.err.println("Config file '" + fileName + "' is malformed");
    }

    return cfg;
  }
```

这个方法会遇到的下面几种异常：

- 配置文件不存在；
- 配置文件在正要读取时消失了；
- 配置文件中有语法错误；
- 配置文件中可能包含无效信息。

这些异常可以分为两大类。一类是文件以某种方式丢失或损坏，另一类是虽然文件理论上存在并且是正确的，却无法正常读取（可能是因为网络或硬件故障）。

如果能把这些异常情况简化为这两类，并且把所有“文件以某种方式丢失或损坏”的异常放在一个catch语句中处理会更好。在Java 7中就可以做到：

代码清单1-2　在Java 7中处理不同的异常

```
public Configuration getConfig(String fileName) {
  Configuration cfg = null;
  try {
    String fileText = getFile(fileName);
    cfg = verifyConfig(parseConfig(fileText));
  } catch (FileNotFoundException|ParseException|ConfigurationException e) {
    System.err.println("Config file '" + fileName +
                       "' is missing or malformed");
  } catch (IOException iox) {
    System.err.println("Error while processing file '" + fileName + "'");
  }

  return cfg;
}
```

异常e的确切类型在编译时还无法得知。这意味着在catch块中只能把它当做可能异常的共同父类（在实际编码时经常用Exception或Throwable）来处理。

另外一个新语法可以为重新抛出异常提供帮助。开发人员经常要在重新抛出异常之前对它进行处理。在前几个版本的Java中，经常可以看到下面这种代码：

```
try {
  doSomethingWhichMightThrowIOException();
  doSomethingElseWhichMightThrowSQLException();
} catch (Exception e) {
  ...
  throw e;
}
```

这会强迫你把新抛出的异常声明为Exception类型——异常的真实类型却被覆盖了。

不管怎样，很容易看出来异常只能是IOException或SQLException。既然你能看出来，编译器当然也能。下面的代码中用了Java 7的语法，只改了一个单词：

```
try {
  doSomethingWhichMightThrowIOException();
  doSomethingElseWhichMightThrowSQLException();
} catch (final Exception e) {
  ...
  throw e;
}
```

关键字final表明实际抛出的异常就是运行时遇到的异常——在上面的代码中就是IOException或SQLException。这被称为final重抛，这样就不会抛出笼统的异常类型，从而避免在上层只能用笼统的catch捕获。

上例中的关键字final不是必需的，但实际上，在向catch和重抛语义调整的过渡阶段，留着它可以给你提个醒。

Java 7对异常处理的改进不仅限于这些通用问题，对于特定的资源管理也有所提升，我们马上就会讲到。

1.3.4　try-with-resources（TWR）

这个修改说起来容易，但其实暗藏玄机，最终证明做起来比最初预想的要难。其基本设想是把资源（比如文件或类似的东西）的作用域限定在代码块内，当程序离开这个代码块时，资源会被自动关闭。

这是一项非常重要的改进，因为没人能在手动关闭资源时做到100%正确，甚至不久前Sun提供的操作指南都是错的。在向Coin项目提交这一提案时，提交者宣称JDK中有三分之二的close()用法都有bug，真是不可思议！

好在编译器可以生成这种学究化、公式化且手工编写易犯错的代码，所以Java 7借助了编译器来实现这项改进。

这可以减少我们编写错误代码的几率。相比之下，想想你用Java 6写段代码，要从一个URL（url）中读取字节流，并把读取到的内容写入到文件（out）中，这么做很容易产生错误。代码清单1-3是可行方案之一。

代码清单1-3　Java 6中的资源管理语法

```
InputStream is = null;
try {
  is = url.openStream();
  OutputStream out = new FileOutputStream(file);
  try {
    byte[] buf = new byte[4096];
    int len;
    while ((len = is.read(buf)) >= 0)
      out.write(buf, 0, len);
  } catch (IOException iox) {
  } finally {
    try {
      out.close();
```


处理异常（能读或写）

```
      } catch (IOException closeOutx) {       <┐
      }                                         │
    }                                           │
  } catch (FileNotFoundException fnfx) {   | 处理异常   │ 遇到异常也
  } catch (IOException openx) {                 │ 做不了什么
  } finally {                                   │
    try {                                       │
      if (is != null) is.close();               │
    } catch (IOException closeInx) {          <┘
    }
  }
```

看明白了吗？重点是在处理外部资源时，墨菲定律（任何事都可能出错）一定会生效，比如：

- URL中的`InputStream`无法打开，不能读取或无法正常关闭；
- `OutputStream`对应的`File`无法打开，无法写入或不能正常关闭；
- 上面的问题同时出现。

最后一种情况是最让人头疼的——异常的各种组合拳打出来令人难以招架。

新语法能大大减少错误发生的可能性，这正是它受欢迎的主要原因。编译器不会犯开发人员编写代码时易犯的错误。

让我们看看代码清单1-3中的代码用Java 7写出来什么样。和前面一样，`url`是一个指向下载目标文件的`URL`对象，`file`是一个保存下载数据的`File`对象。

代码清单1-4　Java 7中的资源管理语法

```
try (OutputStream out = new FileOutputStream(file);
     InputStream is = url.openStream() ) {
  byte[] buf = new byte[4096];
  int len;
  while ((len = is.read(buf)) > 0) {
    out.write(buf, 0, len);
  }
}
```

这是资源自动化管理代码块的基本形式——把资源放在`try`的圆括号内。C#程序员看到这个也许会觉得有点眼熟，是的，它的确很像C#中的从句，带着这种理解使用这个新特性是个不错的起点。在这段代码块中使用的资源在处理完成后会自动关闭。

但使用try-with-resources特性时还是要小心，因为在某些情况下资源可能无法关闭。比如在下面的代码中，如果从文件（`someFile.bin`）创建`ObjectInputStream`时出错，`FileInputStream`可能就无法正确关闭。

```
try ( ObjectInputStream in = new ObjectInputStream(new
      FileInputStream("someFile.bin")) ) {
 ...
}
```

假定文件（`someFile.bin`）存在，但可能不是`ObjectInput`类型的文件，所以文件无法正确打开。因此不能构建`ObjectInputStream`，所以`FileInputStream`也没办法关闭。

要确保try-with-resources生效，正确的用法是为各个资源声明独立变量。

```
try ( FileInputStream fin = new FileInputStream("someFile.bin");
         ObjectInputStream in = new ObjectInputStream(fin) ) {
  ...
}
```

TWR的另一个好处是改善了错误跟踪的能力，能够更准确地跟踪堆栈中的异常。在Java 7之前，处理资源时抛出的异常信息经常会被覆盖。TWR中可能也会出现这种情况，因此Java 7对跟踪堆栈进行了改进，现在开发人员能看到可能会丢失的异常类型信息。

比如在下面这段代码中，有一个返回`InputStream`的值为`null`的方法：

```
try(InputStream i = getNullStream()) {
  i.available();
}
```

在改进后的跟踪堆栈中能看到提示，注意其中被抑制的`NullPointerException`（简称NPE）：

```
Exception in thread "main" java.lang.NullPointerException
  at wgjd.ch01.ScratchSuprExcep.run(ScratchSuprExcep.java:23)
  at wgjd.ch01.ScratchSuprExcep.main(ScratchSuprExcep.java:39)
  Suppressed: java.lang.NullPointerException
  at wgjd.ch01.ScratchSuprExcep.run(ScratchSuprExcep.java:24)
    1 more
```

TWR与`AutoCloseable`

目前TWR特性依靠一个新定义的接口实现`AutoCloseable`。TWR的`try`从句中出现的资源类都必须实现这个接口。Java 7平台中的大多数资源类都被修改过，已经实现了`AutoCloseable`（Java 7中还定义了其父接口`Closeable`），但并不是全部资源相关的类都采用了这项新技术。不过，JDBC 4.1已经具备了这个特性。

然而在你自己的代码里，在需要处理资源时一定要用TWR，从而避免在异常处理时出现bug。

希望你能尽快使用try-with-resources，把那些多余的bug从代码库中赶走。

1.3.5 钻石语法

针对创建泛型定义和实例太过繁琐的问题，Java 7做了一项改进，以减少处理泛型时敲键盘的次数。比如你用`userid`（整型值）标识一些user对象，每个user都对应一个或多个查找表[①]。这用代码应该如何表示呢？

```
Map<Integer, Map<String, String>> usersLists =
    new HashMap<Integer, Map<String, String>>();
```

这简直太长了，并且几乎一半字符都是重复的。如果能写成

① 一种为提高处理速度而用查询取代计算的处理机制。一般是将事先计算好的结果存在数组或映射中，然后在需要该结果时直接读取，比如用三角表查某一角度的正弦值。——译者注

```
Map<Integer, Map<String, String>> usersLists = new HashMap<>();
```

让编译器推断出右侧的类型信息是不是更好？神奇的Coin项目满足了你这个心愿。在Java 7中，像这样的声明缩写完全合法，还可以向后兼容，所以当你需要处理以前的代码时，可以把过去比较繁琐的声明去掉，使用新的类型推断语法，这样可以省出点儿空间来。

编译器为这个特性采用了新的类型推断形式。它能推断出表达式右侧的正确类型，而不是仅仅替换成定义完整类型的文本。

为什么叫“钻石语法”

把它称为”钻石语法”是因为这种类型信息看起来像钻石。原来提案中的名字是“为泛型实例创建而做的类型推断改进”（Improved Type Inference for Generic Instance Creation）。这个名字太长，可缩写ITIGIC听上去又很傻，所以干脆就叫钻石语法了。

新的钻石语法肯定会让你少写些代码。我们最后还要探讨Coin项目中的一个特性——使用变参时的警告信息。

1.3.6 简化变参方法调用

这是所有修改里最简单的一个，只是去掉了方法签名中同时出现变参和泛型时才会出现的类型警告信息。

换句话说，除非你写代码时习惯使用类型为T的不定数量参数，并且要用它们创建集合，否则你就可以进入下一节了。如果你想要写下面这种代码，那就继续阅读本节：

```
public static <T> Collection<T> doSomething(T... entries) {
  ...
}
```

还在？很好。这到底是怎么回事？

变参方法是指参数列表末尾是数量不定但类型相同的参数方法。但你可能还不知道变参方法是如何实现的。基本上，所有出现在末尾的变参都会被放到一个数组中（由编译器自动创建），并作为一个参数传入。

这是个好主意，但是存在一个公认的Java泛型缺陷——不允许创建已知类型的泛型数组。比如下面这段代码，编译就无法通过：

```
HashMap<String, String>[] arrayHm = new HashMap<>[2];
```

不可以创建特定泛型的数组，只能这样写：

```
HashMap<String, String>[] warnHm = new HashMap[2];
```

可这样编译器会给出一个只能忽略的警告。你可以将`warnHm`的类型定义为`HashMap<String, String>`数组，但不能创建这个类型的实例，所以你不得不硬着头皮（或至少忘掉警告）硬生生地把原始类型（`HashMap`数组）的实例塞给`warnHm`。

这两个特性（编译时生成数组的变参方法和已知泛型数组不能是可实例化类型）碰到一起时，会令人有点头疼。看看下面这段代码：

```
HashMap<String, String> hm1 = new HashMap<>();
HashMap<String, String> hm2 = new HashMap<>();

Collection<HashMap<String, String>> coll = doSomething(hm1, hm2);
```

编译器会尝试创建一个包含hm1和hm2的数组，但这种类型的数组应该是被严格禁止使用的。面对这种进退两难的局面，编译器只好违心地创建一个本来不应出现的泛型数组实例，但它又觉得自己不能保持沉默，所以还得嘟囔着警告你这是“未经检查或不安全的操作”。

从类型系统的角度看，这非常合理。但可怜的开发人员本想使用一个十分靠谱的API，一看到这些吓人的警告，却得不到任何解释，不免会内心忐忑。

1. Java 7中的警告去了哪里

Java 7的这个新特性改变了警告的对象。构建这些类型毕竟有破坏类型安全的风险，这总得有人知道。但 API 的用户对此是无能为力的，不管doSomething()是不是干了坏事，破坏了类型安全，都不在API用户的控制范围之内。

真正需要看到这个警告信息的是写doSomething()的人，即API的创建者，而不是使用者。所以Java 7把警告信息从使用API的地方挪到了定义API的地方。

过去是在编译使用API的代码时触发警告，而现在是在编译这种可能会破坏类型安全的API时触发。编译器会警告创建这种API的程序员，让他注意类型系统的安全。

为了减轻API开发人员的负担，Java 7还提供了一个新注解java.lang.SafeVarargs。把这个注解应用到API方法或构造方法之中，则会产生类型警告。通过用@SafeVarargs对这种方法进行注解，开发人员就不会在里面进行任何危险的操作，在这种情况下，编译器就不会再发出警告了。

2. 类型系统的修改

虽然把警告信息从一个地方挪到另一个地方不是改变游戏规则的语言特性，但也证明了我们之前提到的观点——Coin项目曾奉劝诸位贡献者远离类型系统，因为把这么一个小变化讲清楚要大费周章。这个例子表明搞清楚类型系统不同特性之间如何交互是多么费心费力，而且对语言的修改被实现后又会怎么影响这种交互。这还不是特别复杂的修改，更大的变动所涉及的内容还会更多，其中还包括大量微妙的分支。

最后这个例子阐明了由小变化引发的错综复杂的影响。我们对Coin项目中改进的讨论也结束了。尽管它们几乎全都是语法上的小变化，但跟实现它们的代码量相比，它们所带来的正面影响还是很可观的。一旦开始使用，你就会发现这些特性对程序真的很有帮助！

1.4　小结

修改语言非常困难。而用类库实现新特性总是相对容易一些，当然并不是所有特性都能用类库实现。面对挑战时，语言设计师可能会做出一些比他们的预想更轻微、更保守的调整。

现在，我们该去看看构成发布版本更重要的东西了，先从Java 7中某些核心类库的变化开始。我们的下一站是I/O类库，那里可以说是发生了天翻地覆的变化。在此之前，希望你已经掌握了Java之前的版本处理I/O的方法，因为Java 7中的这些类（有时候被称为NIO.2）是构建在之前框架基础之上的。

如果你想看到更多关于TWR实战的例子，或者想要了解最新、高性能的I/O类，那就赶快进入下一章吧！

第2章 新I/O

2

本章内容

- Java 7的新I/O API（即NIO.2）
- `Path`——基于文件和目录的I/O新基础
- `Files`应用类及它的各种辅助方法
- 如何实现常见的I/O应用场景
- 介绍异步I/O

本章重点是Java语言中改变较大的I/O API，被称为“再次更新的I/O”或NIO.2（即JSR-203）。NIO.2是一组新的类和方法，主要存在于`java.nio`包内。下面来看一下它的优点。

- 它完全取代了`java.io.File`与文件系统的交互。
- 它提供了新的异步处理类，让你无需手动配置线程池和其他底层并发控制，便可在后台线程中执行文件和网络I/O操作。
- 它引入了新的`Network-Channel`构造方法，简化了套接字（Socket）与通道的编码工作。

先看案例。老板让你写个程序，要扫描生产服务器上的所有目录，找出曾经用各种读/写和所有者权限写入过的所有properties文件。对于Java 6（及更低版本）而言，这几乎是不可能完成的任务，因为：

- 没有直接支持目录树导航的类或方法；
- 没办法检测和处理符号链接；[①]
- 用简单操作读不出文件的属性（比如可读、可写或可执行）。

用Java 7的NIO.2 API可以完成这个不可能的编程任务，它支持目录树的直接导航（`Files.walkFileTree()`，2.3.1节）、符号链接（`Files.isSymbolicLink()`，代码清单2-4），能用一行代码读取文件属性（`Files.readAttributes()`，2.4.3节）。

除此之外，老板还要求你在读取这些properties文件时不能打断主程序的处理流程。可最小的properties文件也有1MB，读取这些文件很可能打断程序的主流程！面对这一要求，在Java 5/6的时代，你很可能会用`java.util.concurrent`包中的类创建线程池和工作线程队列，再用单独

① 符号链接是一种特殊类型的文件，指向文件系统中的另外一个文件或位置——你可以把它理解为快捷方式。

的后台线程读取文件。我们在第4章将会讨论到，现在Java中的并发仍然相当困难，并且非常容易出错。借助Java 7和NIO.2 API，你可以用新的`AsynchronousFileChannel`（2.5节），不用指定工作线程或队列就可以在后台读取大型文件。咻！

这个新API非常有用，尽管它不能帮你冲咖啡，但它的发展趋势可在那儿摆着呢。

第一个趋势是对其他数据存储方法的探索，特别是在非关系或大数据集领域。你可能很快就会遇到读写大文件（比如微博上的大型报告文件）的问题。NIO.2可以帮助你用一种异步、有效的方式读写大文件，还能利用底层操作系统的特性。

第二个趋势是多核CPU的发展，使得真正并发且更快的I/O成为可能。并发是个难以掌握的领域[①]，但NIO.2会助你一臂之力，它为多线程文件和套接字访问的应用提供了一个简单的抽象层。即便你不直接使用这些特性，它们也会对你的编程生涯产生极大影响，因为IDE、应用服务器和各种流行的框架会大量应用这些特性。

这些只是NIO.2会对你有哪些帮助的例子。如果NIO.2可以解决你眼下面临的一些问题，本章的内容就是为你准备的！否则，你可以在接到Java I/O 任务时再回来。

本章你会体验到Java 7新I/O的能力，以便你能够开始编写基于NIO.2的代码，并有信心探索新的API。除此之外，这些API还使用了一些第1章提到的特性，这证明Java 7确实会使用自己的特性。

提示　将try-with-resources和NIO.2中的新API结合起来可以写出非常安全的I/O程序，这在Java中还是破天荒的第一次！

我们觉得你很可能会用到新的文件I/O能力，所以本章会非常详细地介绍。你需要从了解新的文件系统抽象层开始，即先了解`Path`和它的辅助类。在`Path`之上，你会接触到常用的文件系统操作，比如复制和移动文件。

我们还会向你介绍异步I/O，给你看一个文件系统的例子。最后我们会讨论套接字和通道功能的融合，以及这对于网络应用开发人员意味着什么。但我们先来看一下NIO.2的由来。

2.1　Java I/O 简史

要品出NIO.2 API设计的真正味道，并深刻理解它们的用法，应该先弄清Java I/O的历史。但我们非常理解你想要接触代码的渴望。别着急，你的这种渴望可以在2.2节得到满足。

如果你发现某些API的用法非常简单甚至有点古怪，本节会帮助你从API设计者的角度看待NIO.2。这是Java I/O的第三个主要版本，所以让我们回顾一下Java对I/O支持的发展历史，看看NIO.2是怎么产生的。

Java之所以能够广泛流传，其强大、丰富、简明的类库功不可没，编程时要解决的大多数问

---

① 第4章深入探讨了并发计算可能给你的编程生涯带来的微妙复杂性。

题几乎都可以在其中找到支持。但经验丰富的Java开发人员都知道，在老版本的Java中，有些地方不是那么给力。曾经让他们最崩溃的就是Java的输入/输出（I/O）API。

2.1.1 Java 1.0 到 1.3

在Java的早期版本（1.0~1.3）中没有完整的I/O支持。也就是说开发人员在开发需要I/O支持的应用时，要面临如下问题。

- 没有数据缓冲区或通道的概念，开发人员要编程处理很多底层细节。
- I/O操作会被阻塞，扩展能力受限。
- 所支持的字符集编码有限，需要进行很多手工编码工作来支持特定类型的硬件。
- 不支持正则表达式，数据处理困难。

基本上，早期版本的Java缺乏对非阻塞I/O的支持。开发人员万般无奈，只好自己实现可伸缩的I/O解决方案。在Java 1.4发布之前，Java一直没能在服务器端开发领域得到重用，我们认为主要原因就是缺乏对非阻塞I/O的支持。

2.1.2 在Java 1.4 中引入的NIO

为了解决这些问题，Java开始实现对非阻塞I/O的支持，与其他I/O特性一起，帮助开发人员交付更快、更可靠的I/O解决方案。其中有两次主要改进：

- 在Java 1.4中引入非阻塞I/O；
- 在Java 7中对非阻塞I/O进行修改。

2002年发布Java 1.4时，非阻塞I/O（NIO）以JSR-51的身份加入到Java语言中。下面这些特性就是那时增加的。自此之后，Java语言终于得到了服务器端开发人员的青睐：

- 为I/O操作抽象出缓冲区和通道层；
- 字符集的编码和解码能力；
- 提供了能够将文件映射为内存数据的接口；
- 实现非阻塞I/O的能力；
- 基于流行的Perl实现的正则表达式类库。

Perl——正则表达式之王

毋庸置疑，Perl编程语言是正则表达式处理之王。实际上，它的设计和实现相当出色，甚至很多编程语言（包括Java）竞相模仿Perl的语法和语义。如果你对Perl语言感兴趣，可以访问http://www.perl.org/一探究竟。

NIO无疑使Java向前迈出了一大步，但I/O编程对Java开发人员来说仍然是个挑战。特别是对于文件系统中的文件和目录处理而言，支持力度还是不够。那时的`java.io.File`类有些比较烦人的局限性。

- 在不同的平台中对文件名的处理不一致。[①]
- 没有统一的文件属性模型。（比如读写访问模型）
- 遍历目录困难。
- 不能使用平台/操作系统的特性。[②]
- 不支持文件系统的非阻塞操作。[③]

2.1.3 下一代I/O-NIO.2

为了突破这些局限性，同时也为了支持现代硬件和软件I/O的新范例，由阿兰•波特曼主导的JSR-203应运而生。JSR-203最终变成了我们在Java 7中见到的NIO.2 API。它有三个主要目标，JSR-203规范中的第2.1节对它们进行了详细的介绍（http://jcp.org/en/jsr/detail?id=203）。

(1) 一个能批量获取文件属性的文件系统接口，去掉和特定文件系统相关的API，还有一个用于引入标准文件系统实现的服务提供者接口。

(2) 提供一个套接字和文件都能够进行异步（与轮询、非阻塞相对）I/O操作的API。

(3) 完成JSR-51中定义的套接字——通道功能，包括额外对绑定、选项配置和多播数据报的支持。

接下来让我们从新文件系统的基础`Path`和它的朋友们开始吧！

2.2 文件 I/O 的基石：Path

在NIO.2的文件I/O中，`Path`是必须掌握的关键类之一。`Path`通常代表文件系统中的位置，比如C:\workspace\java7developer（Windows文件系统中的目录）或/usr/bin/zip（*nix文件系统中zip程序的位置）。如果你能理解如何创建和处理路径，就能浏览任何类型的文件系统，包括zip归档文件系统。

我们通过图2-1（基于本书中的源码布局）来复习一下文件系统中的几个概念。

- 目录树
- 根路径
- 绝对路径
- 相对路径

之所以要讨论绝对路径和相对路径，是因为我们需要考虑程序运行的位置。比如说，有个程序可能在/java7developer/src/test目录下运行，代码要读取位于/java7developer/src/main目录下的文件名。为了进入/java7developer/src/main目录，可以用相对路径../main。

但如果程序运行在/java7developer/src/test/java/com/java7developer，用相对路径 ../main则无法

① 一些批评者会说Java没有兑现“一次编写，处处运行”的承诺。
② 人们通常希望能够使用Linux/UNIX中的符号链接机制。
③ Java 1.4的确支持网络套接字的非阻塞操作。

进入目标目录，而会进到并不存在的目录/java7developer/src/test/java/com/main中。所以必须使用绝对路径，比如/java7developer/src/main。

反之亦然，你的程序可能一直运行在同一位置，比如图2-1中的target目录。但目录树的根目录可能会变，比如从/java7developer变成了D:\workspace\j7d。这时就不能依靠绝对路径，而要用相对路径转到你想到达的位置。

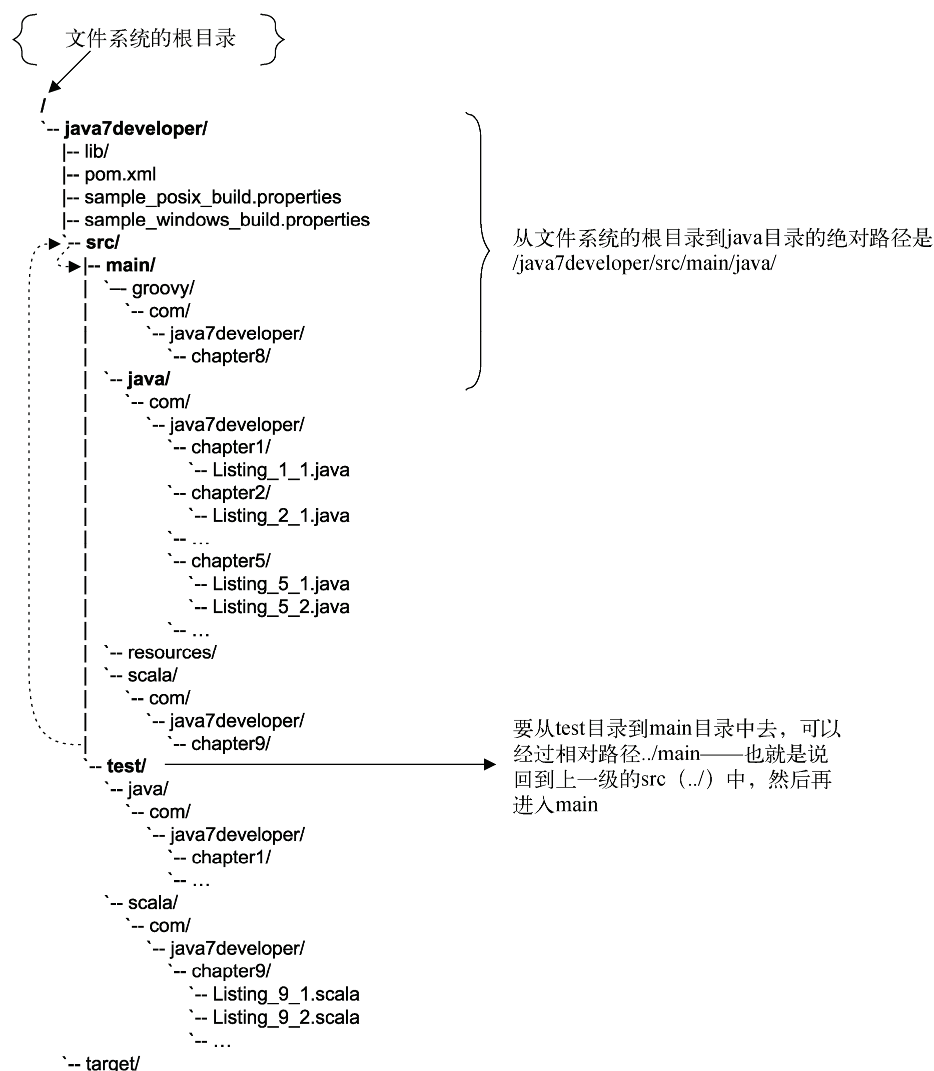


图2-1　说明根目录、绝对路径和相对路径概念的目录树

NIO.2中的`Path`是一个抽象构造。你所创建和处理的`Path`可以不马上绑定到对应的物理位置上。尽管看起来奇怪，但有时确实需要如此。比如说，你想创建一个`Path`来表示即将创建的新

文件。在调用`Files.createFile(Path target)`[1]之前，这个文件是不存在的。如果在`Path`所对应的文件创建之前，你试图读取这个文件中的内容，就会导致`IOException`。如果你指定了一个并不存在的`Path`并试图用`Files.readAllBytes(Path)`之类的方法读取，结果是一样的。简言之，JVM只会把`Path`绑定到运行时的物理位置上。

警告　在编写与具体文件系统相关的代码时要小心。创建`Path`为C:\workspace\java7developer，然后试图读取它的程序，这只能在有C:\workspace\java7developer位置的计算机上才能工作。一定要确保程序逻辑和异常处理考虑到了各种情况，包括可能在不同的文件系统上运行，或文件系统的结构可能被改动。本书的作者之一过去就犯过这个错误，结果把计算机系的一组硬盘全搞坏了！[2]

还是得再重复一遍，NIO.2把位置（由`Path`表示）的概念和物理文件系统的处理（比如复制一个文件）分得很清楚，物理文件系统的处理通常是由`Files`辅助类实现的。

有关`Path`类以及将在本节讨论的其他类的进一步细节请见表2-1。

表2-1　学习文件I/O的关键基础类

类	说　明
`Path`	`Path`类中的方法可以用来获取路径信息，访问该路径中的各元素，将路径转换为其他形式，或提取路径中的一部分。有的方法还可以匹配路径字串以及移除路径中的冗余项
`Paths`	工具类，提供返回一个路径的辅助方法，比如`get(String first, String... more)`和`get(URI uri)`
`FileSystem`	与文件系统交互的类，无论是默认的文件系统，还是通过其统一资源标识（URI）获取的可选文件系统
`FileSystems`	工具类，提供各种方法，比如其中用于返回默认文件系统的`FileSystems.getDefault()`

记住，`Path`不一定代表真实的文件或目录。你可以随心所欲地操作`Path`，用`Files`中的功能来检查文件是否存在，并对它进行处理。

提示　`Path`并不仅限于传统的文件系统，它也能表示zip或jar这样的文件系统。

我们来完成几个简单的任务，探索一下`Path`类：

- ❑ 创建一个`Path`；
- ❑ 获取`Path`的相关信息；
- ❑ 移除`Path`中的冗余项；

---

① 你一会儿就能在2.4节见到这个方法！

② 我们不会告诉你是谁，不过欢迎你把他查出来！

- 转换一个Path；
- 合并两个Path，在两个Path中间创建一个Path，并对这两个Path进行比较。

我们先从创建表示文件系统中某个位置的Path开始。

2.2.1　创建一个Path

创建Path没什么难的。调用Paths.get(String first,String...more)是最快捷的做法，其中第二个变量一般用不到，它仅仅用来把额外的字符串合并起来形成Path字符串。

> 提示　在NIO.2的API中，Path或Paths中的各种方法抛出的受检异常只有IOException。我们认为这虽然简单，但有时却会掩藏潜在的问题，而且如果你想处理IOException的某个显式子类，则需要额外编写异常处理代码。

我们用Paths.get(String first)方法为/usr/bin/目录下的文件压缩工具zip创建一个绝对Path。

```
Path listing = Paths.get("/usr/bin/zip");
```

调用Paths.get("/usr/bin/zip")的效果和调用下面这个长一点的代码效果一样：

```
Path listing = FileSystems.getDefault().getPath("/usr/bin/zip");
```

> 提示　创建Path时可以用相对路径。比如，运行在/opt目录下的程序可以用../usr/bin/zip创建一个指向/usr/bin/zip的Path。其含义是指进入/opt的上一层目录（即/），然后进入/usr/bin/zip。通过调用toAbsolutePath()方法，很容易把这个相对路径转换成绝对路径，如：listing.toAbsolutePath()。

你可以从Path中获取信息，比如其父目录、文件名（如果有的话）等。

2.2.2　从Path中获取信息

Path类中有一组方法可以返回你正在处理的路径的相关信息。代码清单2-1为/usr/bin目录下的zip工具创建一个Path，并输出相关信息，包括它的根目录和父目录。如果你的操作系统中zip也放在/usr/bin目录下，应该能见到下面这种输出信息。

```
File Name [zip]
Number of Name Elements in the Path [3]
Parent Path [/usr/bin]
Root of Path [/]
Subpath from Root, 2 elements deep [usr/bin]
```

你在计算机上看到的结果和你所用的操作系统及程序运行的位置有关。

代码清单2-1　从Path中获取信息

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class Listing_2_1 {

  public static void main(String[] args) {
    Path listing = Paths.get("/usr/bin/zip");                    <- 创建绝对路径
    System.out.println("File Name [" +
                       listing.getFileName() + "]");              <- 获取文件名
    System.out.println("Number of Name Elements
                       in the Path [" +
                       listing.getNameCount() + "]");             ❶ 获取名称元素的数量
    System.out.println("Parent Path [" +
                       listing.getParent() + "]");
    System.out.println("Root of Path [" +
                       listing.getRoot() + "]");                  ❷ 获取Path的信息
    System.out.println("Subpath from Root,
                       2 elements deep [" +
                       listing.subpath(0, 2) + "]");

  }
}
```

在创建了/usr/bin/zip的Path之后，你可以看一下组成Path的元素个数，在此例中就是目录的数量❶。相对其父目录和根目录，Path的位置会更有用。也可以通过指定起始和终止的索引来挑出子路径。在本例中是从Path的根（0）到其第二个元素（2）之间的子路径❷。

如果这是你初次接触NIO.2文件API，这些方法对你来说非常重要，因为你可以借助它们查看路径处理的结果。

2.2.3　移除冗余项

在编写工具软件，比如属性文件分析器时，需要处理的Path中可能会有一个或两个点：

- . 表示当前目录；
- .. 表示父目录。

假设你的程序在/java7developer/src/main/java/com/java7developer/chapter2/目录下运行（见图2-1）。你所在的目录和Listing_2_1.java一样，如果传给你的Path是./Listing_2.1.java，其中的./部分，即程序正在运行的目录，实际上并没什么用。在这里用短一点儿的Path——Listing_2_1.java就够了。

Path中可能还有其他冗余项，比如符号链接（见2.4.3节）。假设在*nix系统中/usr/logs目录下，你想寻找日志文件log1.txt，但其实/usr/logs只是一个指向/application/logs的符号链接，那里才是存放日志文件的真正位置。要得到这个位置，就需要去掉冗余的符号信息。

所有这些冗余项都会导致Path指向的不是你认为它应该指向的位置。

在Java 7中，有两个辅助方法可以用来弄清Path的真实位置。首先可以用normalize()方法去掉Path中的冗余信息。下面的代码会返回Listing_2_1.java的Path，去掉了表明它在当前目录（./部分）中的冗余符号。

```
Path normalizedPath = Paths.get("./Listing_2_1.java").normalize();
```

此外，toRealPath()方法也很有效，它融合了toAbsolutePath()和normalize()两个方法的功能，还能检测并跟随符号连接。

还是回到*nix系统中的日志文件那个例子，在/usr/logs目录下有个日志文件log1.txt，而这个目录实际上是指向/application/logs的符号链接。通过调用toRealPath()，你能得到表示/application/logs/log1.txt的真正Path。

```
Path realPath = Paths.get("/usr/logs/log1.txt").toRealPath();
```

我们要讨论的最后一个Path API特性是比较多个Path，并找出它们之间的Path。

2.2.4　转换**Path**

转换路径最多的是工具软件。比如你可能需要比较文件之间的关系，以便了解源码目录树的结构是否符合编码规范。或者在shell脚本中执行程序时可能会传入一些Path参数，并需要把这些参数变成有意义的Path。在NIO.2里可以很容易地合并Path，在两个Path中再创建Path或对Path进行比较。

下面的代码将两个Path合并，通过调用resolve方法，将uat和conf/application.properties合并成表示/uat/conf/application.properties的完整Path。

```
Path prefix = Paths.get("/uat/");
Path completePath = prefix.resolve("conf/application.properties");
```

要取得两个Path之间的路径，可以用relativize(Path)方法。下面的代码计算了从日志目录到配置目录之间的路径。

```
String logging = args[0];
String configuration = args[1];
Path logDir = Paths.get(logging);
Path confDir = Paths.get(configuration);
Path pathToConfDir = logDir.relativize(confDir);
```

如你所愿，你可以用startsWith(Path prefix)、equals(Path path)等值比较或endsWith(Path suffix)来对路径进行比较。

现在使用Path类对你来说应该没有问题了，但Java 7之前版本的那些代码怎么办呢？负责NIO.2的团队考虑到了向后兼容性，所以加了两个新的API特性来保证基于Path的新I/O和老版本代码之间的互操作性。

2.2.5　NIO.2 **Path**和Java已有的**File**类

新API中的类可以完全替代过去基于java.io.File的API。但你不可避免地要和大量遗留下来的I/O代码交互。Java 7提供了两个新方法：

- java.io.File类中新增了toPath()方法，它可以马上把已有的File转化为新的Path。
- Path类中有个toFile()方法，它可以马上把已有的Path转化为File。

下面的代码演示了这一功能。

```
File file = new File("../Listing_2_1.java");
Path listing = file.toPath();
listing.toAbsolutePath();
file = listing.toFile();
```

现在，我们完成了对Path类的探索。接着要研究一下Java 7对目录处理的支持，特别是对目录树。

2.3　处理目录和目录树

在读2.2节关于路径的内容时，你可能已经猜到目录不过是带有特别属性的Path。遍历目录的能力是Java 7引人注目的新特性。新加入的java.nio.file.DirectoryStream<T>接口和它的实现类提供了很多功能：

- 循环遍历目录中的子项，比如查找目录中的文件；
- 用glob表达式[①]（比如*Foobar*）进行目录子项匹配和基于MIME的内容检测（比如text/xml文件）；
- 用walkFileTree方法实现递归移动、复制和删除操作。

本节主要讨论两个常见用例：在一个目录中查找文件以及在目录树中执行相同的任务。我们先从最简单的开始：在一个目录中查找任意文件。

2.3.1　在目录中查找文件

我们先讨论一个简单的例子，用模式匹配过滤出java7developer项目中所有的.properties文件。请看下面的代码：

代码清单2-2　列出目录下的properties文件

```
Path dir = Paths.get("C:\\workspace\\java7developer");    ❶ 设定起始路径

try(DirectoryStream<Path> stream
      = Files.newDirectoryStream(dir, "*.properties")) {    ❷ 声明过滤流
  for (Path entry: stream)
  {
    System.out.println(entry.getFileName());    ❸ 找出所有.properties文件并输出
  }
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```

最前面是我们已经熟悉的Paths.get(String)调用❶。紧随其后的是关键方法Direct-

① glob通常指有限的模式匹配，源自早期Unix中的glob()库函数，该函数用于查找文件系统中指定模式的路径，虽然所用语法和正则表达式类似，但没有正则表达式那么强大的表达力，详见附录B。——译者注

oryStream(Path directory, String patternMatch)❷，它返回一个经过过滤的DirectoryStream，其中包含以.properties结尾的文件。最后输出每个子项❸。

过滤流中用到的模式匹配称为glob模式匹配，它和Perl正则表达式类似，但稍有不同。附录B中有如何使用glob模式匹配的详细说明。

代码清单2-2展示了新API处理单个目录的能力，但如果需要递归过滤多个目录时该怎么办？

2.3.2 遍历目录树

Java 7支持整个目录树的遍历。也就是说你可以很容易地搜寻目录树中的文件，在子目录中查找，并对它们执行操作。比如你可能想在做开发的机器上弄个工具类来删除目录/opt/workspace/java下的所有.class文件，完成构建前的清除工作。

遍历目录树是Java 7的新特性，要想正确使用，你得掌握一些接口及其实现的细节。其中的关键方法是：

```
Files.walkFileTree(Path startingDir, FileVisitor<? super Path> visitor);
```

提供startingDir非常简单，但给出FileVisitor接口的实现类就比较麻烦了（参数FileVisitor<? superPath> visitor看上去就不是善茬儿），因为最少得实现下面5个方法（T一般就是Path）：

- FileVisitResult preVisitDirectory(T dir)
- FileVisitResult preVisitDirectoryFailed(T dir, IOException exc)
- FileVisitResult visitFile(T file, BasicFileAttributes attrs)
- FileVisitResult visitFileFailed(T file, IOException exc)
- FileVisitResult postVisitDirectory(T dir, IOException exc)

看起来挺吓人的吧？好在Java 7 API的设计者们已经提供了一个默认实现类，SimpleFileVisitor<T>。

我们要扩展并修改代码清单2-2。下面的代码会列出C:\workspace\java7developer\src目录下及其子目录内的所有.java文件。这段代码展示了Files.walkFileTree方法对默认实现类SimpleFileVisitor的用法，用一个特定的visitFile方法实现来改进它。

代码清单2-3 列出子目录下的所有java源码文件

```
public class Find
{

  public static void main(String[] args) throws IOException
  {
    Path startingDir =
      Paths.get("C:\\workspace\\java7developer\\src");      // 设置起始目录
    Files.walkFileTree(startingDir,
                       new FindJavaVisitor());               // ❶ 调用walkFileTree
  }

  private static class FindJavaVisitor
```

```
                        extends SimpleFileVisitor<Path>    ❷ 扩展SimpleFile-Visitor <Path>
  {

    @Override
    public FileVisitResult
      visitFile(Path file, BasicFileAttributes attrs)    ❸ 重写visitFile方法
    {
      if (file.toString().endsWith(".java")) {
        System.out.println(file.getFileName());
      }
      return FileVisitResult.CONTINUE;
    }
  }
}
```

整个过程从调用`Files.walkFileTree`方法开始❶。这里的关键是`FindJavaVisitor`，该类扩展了`FileVisitor`的默认实现类`SimpleFileVisitor`❷。你想让`SimpleFileVisitor`来完成大部分工作，比如遍历目录。可实际上你唯一要做的就是重写`visitFile(Path, BasicFileAttributes)`[①]方法❸，在这个方法中你也只需要写些代码来判断文件名是否以.java结尾，如果确实是，就在stdout中输出。

其他用例包括递归移动、复制、删除或者修改文件。在大多数应用场景中，你只需要扩展`SimpleFileVisitor`。但如果你想实现自己的`FileVisitor`，API也很灵活。

注意　为了确保递归等操作的安全性，`walkFileTree`方法不会自动跟随符号链接。如果你确实需要跟随符号链接，就需要检查那个属性（如2.4.3节所述）并执行相应的操作。

现在你对路径和目录树已经熟悉了，该从处理位置进入真正的文件系统操作上了，接下来我们会向你介绍新的`Files`类及它的朋友们。

2.4　NIO.2 的文件系统 I/O

对于文件系统的操作任务，比如移动文件、修改文件属性，以及处理文件内容等，在NIO.2中都有所改善。对这些操作的支持主要是由`Files`类提供的。

表2-2中有关于`Files`类的详细介绍，此外本节还会介绍另外一个也很重要的类：`WatchService`。

表2-2　文件处理的基础类

类	说　明
`Files`	让你轻松复制、移动、删除或处理文件的工具类，有你需要的所有方法
`WatchService`	用来监视文件或目录的核心类，不管它们有没有变化

① 2.4节会介绍`BasicFileAttributes`，所以暂时不用管它。

在本节中，你将学会如何在文件和文件系统上执行下面这些任务：

- ❑ 创建和删除文件；
- ❑ 移动、复制、重命名和删除文件；
- ❑ 文件属性的读写；
- ❑ 文件内容的读取和写入；
- ❑ 处理符号链接；
- ❑ 用`WatchService`发出文件修改通知；
- ❑ 使用`SeekableByteChannel`——一个可以指定位置及大小的增强型字节通道。

这看起来可能挺恐怖的，但由于设计巧妙，API提供了很多辅助方法，把抽象层隐藏了起来，让你可以轻松快捷地处理文件系统。

警告 NIO.2 API对原子操作的支持有很大改进，但涉及文件系统处理时，仍然主要依靠代码来提供保护。即使是执行了一半的操作，也很可能会因为突然断网、咖啡泼到服务器上，或某个冒失鬼在错误的UNIX机器上执行了`shutdown now`命令（本书作者之一亲身经历的著名事件）等诸多原因而出错。尽管API的某些方法还是会偶尔抛出个`RuntimeException`，但某些异常状况可以由`Files.exists(Path)`这样的辅助方法来缓解。

学习新API最好的办法就是读写代码。接下来我们来看一些实际案例，先从基本的文件创建和删除开始。

2.4.1 创建和删除文件

只需要调用`Files`类里的辅助方法，就可以很容易地创建和删除文件。当然，你接到的任务不可能总像默认情况那么简单，所以我们额外加了一些选项，比如在新创建的文件上设定可读/可写/可执行的安全访问权限。

提示 如果你要在自己的机器上运行本节中的代码，请用实际路径替换掉代码中的路径。

下面的代码展示了基本的文件创建操作，用到了`Files.createFile(Path target)`方法。如果你的操作系统里有个D:\Backup目录，运行代码之后就会在那里创建一个MyStuff.txt文件。

```
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Path file = Files.createFile(target);
```

通常出于安全考虑，要定义所创建的文件是用于读、写、执行，或三者权限的某种组合时，你要指明该文件的某些`FileAttributes`。因为这取决于文件系统，所以需要使用与文件系统相关的文件权限类。

下面是一个在POSIX文件系统[1]上为属主、属主组内用户和所有用户设置读/写许可的例子。这种方法允许所有用户对即将创建的文件D:\Backup\MyStuff.txt进行读写操作。

```
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rw-rw-rw-");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.createFile(target, attr);
```

在java.nio.file.attribute包里有一大串已经写好的*FilePermission类。对文件属性的支持在2.4.3节中还有更详细的论述。

警告　如果在创建文件时要指定访问许可，不要忽略其父目录强加给该文件的umask限制或受限许可。比如说，你会发现即便你为新文件指定了rw-rw-rw许可，但由于目录的掩码，实际上文件最终的访问许可却是rw-r--r--。

删除文件要简单一些，可以用Files.delete(Path)方法。下面的代码删除了刚刚创建的D:\Backup\MyStuff.txt文件。当然，运行这个Java程序的用户需要有删除文件的权限。

```
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.delete(target);
```

接下来你将学到如何在文件系统中复制和移动文件。

2.4.2　文件的复制和移动

使用Files类中简单的辅助方法可以很轻松地完成文件的复制和移动。

下面的代码演示了如何用Files.copy(Path source, Path target)方法完成基本的复制操作。

```
Path source = Paths.get("C:\\My Documents\\Stuff.txt");
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.copy(source, target);
```

复制文件时通常需要设置某些选项。下面这个例子用到了覆盖即替换已有文件的选项。

```
import static java.nio.file.StandardCopyOption.*;

Path source = Paths.get("C:\\My Documents\\Stuff.txt");
Path target = Paths.get("D:\\Backup\\MyStuff.txt");
Files.copy(source, target, REPLACE_EXISTING);
```

其他的复制选项包括COPY_ATTRIBUTES（复制文件属性）和ATOMIC_MOVE（确保在两边的操作都成功，否则回滚）。

移动和复制很像，都是用原子Files.move(Path source, Path target)方法完成的。通常在移动文件时，你想要用复制选项，此时便可以用Files.move(Path source, Path

---

① 可移植操作系统接口（UNIX），是一种许多操作系统都支持的基本标准。

target, CopyOptions...)方法，但要注意变参的使用。

在下面这个例子中，我们要在移动源文件时保留其属性，并且覆盖目标文件（如果存在的话）。

```
import static java.nio.file.StandardCopyOption.*;

Path source = Paths.get("C:\\My Documents\\Stuff.txt");
Path target = Paths.get("D:\\Backup\\MyStuff.txt");

Files.move(source, target, REPLACE_EXISTING, COPY_ATTRIBUTES);
```

现在你已经能创建、删除、复制和移动文件了，下面该认真研究一下Java 7对文件属性的支持了。

2.4.3 文件的属性

文件属性控制着谁能对文件做什么。一般情况下，做什么许可包括能否读取、写入或执行文件，而由谁许可包括属主、群组或所有人。

本节从讨论文件的基本属性组开始，比如文件最后访问时间以及它是目录还是符号链接等。本节的第二部分讨论对特定文件系统的文件属性的支持，因为不同的文件系统都有它们自己的属性集和属性含义的解释，所以这部分比较难。

让我们先从了解Java 7 对基本文件属性的支持开始吧。

1. 基本文件属性支持

真正通用的文件属性并不多，但确实有一组大多数文件系统都支持的属性。接口BasicFileAttributes定义了这个通用集，但实际上工具类Files就可以回答与文件相关的各种问题，比如下面这些：

- 最后修改时间是什么时候？
- 它有多大？
- 它是符号连接吗？
- 它是目录吗？

代码清单2-4说明了Files类中用于收集这些基本文件属性的方法。代码输出了/usr/bin/zip的相关信息，你看到的输出应该和下面的类似：

```
/usr/bin/zip
2011-07-20T16:50:18Z
351872
false
false
{lastModifiedTime=2011-07-20T16:50:18Z,
fileKey=(dev=e000002,ino=30871217), isDirectory=false,
lastAccessTime=2011-06-13T23:31:11Z, isOther=false,
isSymbolicLink=false, isRegularFile=true,
creationTime=2011-07-20T16:50:18Z, size=351872}
```

注意，所有这些属性都是调用Files.readAttributes(Path path, Stringattributes, LinkOption... options)得到的。代码清单2-4如下所示：

代码清单2-4　通用的文件属性

```
try
{
  Path zip = Paths.get("/usr/bin/zip");
  System.out.println(Files.getLastModifiedTime(zip));
  System.out.println(Files.size(zip));
  System.out.println(Files.isSymbolicLink(zip));
  System.out.println(Files.isDirectory(zip));
  System.out.println(Files.readAttributes(zip, "*"));
}
catch (IOException ex)
{
  System.out.println("Exception [" + ex.getMessage() + "]");
}
```

获取Path

输出属性

执行批量读取

还有一些可以从Files类的方法中采集到的通用文件属性信息。这样的信息包括文件属主，是否为符号链接等。请参照Files类的Javadoc查看完整的辅助方法列表。

Java 7也支持跨文件系统的文件属性查看和处理功能。

2. 特定文件属性支持

在2.4.1节创建文件时你已经见过FileAttribute接口和PosixFilePermissions类了。为了支持文件系统特定的文件属性，Java 7允许文件系统提供者实现FileAttributeView和BasicFileAttributes接口。

警告　我们之前已经说过了，但有必要再重复一次。在编写特定文件系统的代码时一定要小心。一定要确保你的逻辑和异常处理考虑到了代码在不同文件系统上运行的情况。

来看一个例子，其中你想用Java 7保证正确的访问许可被设置在特定文件中。图2-2显示了Admin用户的home目录的列表。注意那个特殊的.profile隐藏文件，它只允许Admin用户写，其他任何人都没有写权限，但所有人都可以读取该文件。

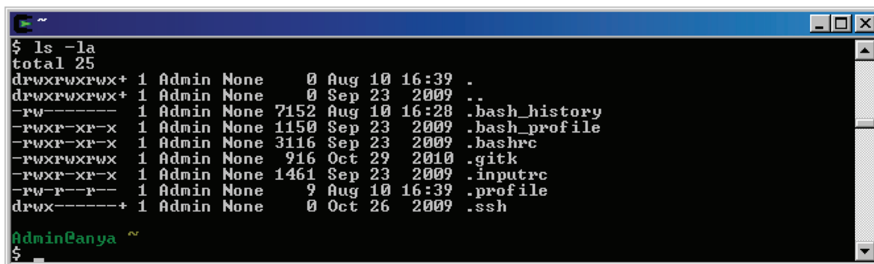


图2-2　Admin用户的home目录列表，显示.profile的访问许可

在下面的代码中，你要确保.profile文件的访问许可设置正确，与图2-2对应。Admin用户希望其他所有用户都可以读取该文件，但只有他自己来写。你可以用特定的POSIX PosixFilePermission和PosixFileAttributes类来保证访问许可（rw-r--r--）是正确的。

代码清单2-5 Java 7对文件属性的支持

```
import static java.nio.file.attribute.PosixFilePermission.*;

try
{
  Path profile = Paths.get("/user/Admin/.profile");

  PosixFileAttributes attrs =
      Files.readAttributes(profile,
                           PosixFileAttributes.class);     ❶ 获取属性视图

  Set<PosixFilePermission> posixPermissions =
                                attrs.permissions();      ❷ 读取访问许可
  posixPermissions.clear();                                ❸ 消除访问许可

  String owner = attrs.owner().getName();
  String perms =
      PosixFilePermissions.toString(posixPermissions);
  System.out.format("%s %s%n", owner, perms);              日志信息

  posixPermissions.add(OWNER_READ);
  posixPermissions.add(GROUP_READ);
  posixPermissions.add(OTHER_READ);
  posixPermissions.add(OWNER_WRITE);
  Files.setPosixFilePermissions(profile, posixPermissions);  ❹ 设置新的访问许可
}
catch(IOException e)
{
  System.out.println(e.getMessage());
}
```

代码从导入`PosixFilePermission`常量还有其他未显示的导入开始，然后得到.profile文件的`Path`。`Files`类中有个辅助方法让你可以读取特定文件系统的属性，在这个例子中是`PosixFileAttributes`❶。然后你就可以访问`PosixFilePermission`❷。在清除了已有的许可之后❸，你可以为文件添加新的访问许可，当然还是用`Files`中的方法❹。

你可能已经注意到了，`PosixFilePermission`是一个`enum`，因此没有实现`FileAttributeView`接口。为什么这里没用`PosixFileAttributeView`呢？实际上是`Files`辅助类把它隐藏了起来，这样你就可以用`readAttributes`方法直接读取文件属性了，也可以用`setPosixFilePermissions`方法直接设置访问许可。

除了基本属性，Java 7还有一个用来支持特别操作系统特性的扩展系统。可惜，我们不可能囊括所有特殊情况，但我们会给你看一个扩展系统的例子：Java 7对符号链接的支持。

3. 符号链接

你可以把符号链接看做指向另一个文件或目录的入口，并且在大多数情况下它们都是透明的。比如切换到符号链接的目录下会把你带到符号链接所指向的目录下。但在写软件时，比如备份工具或部署脚本，你需要慎重考虑是否应该跟随符号链接，NIO.2允许你做出选择。

我们再用一下2.2.3节的例子。你要在*nix系统上查询/usr/logs目录下的日志文件log1.txt的信息。但/usr/logs目录实际上是一个指向/application/logs目录的符号链接（指针），/application/logs目录才是日志文件的真正位置。

符号链接在宿主操作系统中使用，包括（但不限于）UNIX、Linux、Windows 7和Mac OS X。Java 7对符号链接的支持遵循UNIX操作系统中实现的语义。

下面的代码在读取基本文件属性之前先检查指向安装Java的/opt/platform目录的`Path`，看它是否为符号链接，我们想读取文件真正位置的属性。代码清单2-6如下所示：

代码清单2-6　探索符号链接

```
Path file = Paths.get("/opt/platform/java");
try
{
  if(Files.isSymbolicLink(file))
  {
    file = Files.readSymbolicLink(file);
  }
  Files.readAttributes(file, BasicFileAttributes.class);
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```

❶ 检查符号链接
❷ 读取符号链接
❸ 读取文件属性

`Files`类提供了一个`isSymbolicLink(Path)`方法来检查符号链接❶。它还有一个辅助方法，可以用于返回符号链接目标的真实`Path`❷，所以你能读到正确的文件属性❸。

NIO.2 API默认会跟随符号链接。如果不想跟随，需要用`LinkOption.NOFOLLOW_LINKS`选项。这一选项可以用在几个方法调用上。如果你要读取符号链接本身的基本文件属性，应该调用：

```
Files.readAttributes(target,
                     BasicFileAttributes.class,
                     LinkOption.NOFOLLOW_LINKS);
```

符号链接是Java 7对特定文件系统支持最常用的例子，API设计者也考虑到了未来对特定文件系统支持特性的扩展，比如量子加密文件系统。

你已经做过文件处理了，现在可以开始研究对文件内容的处理了。

2.4.4　快速读写数据

Java 7可以尽可能多地提供用来读取和写入文件内容的辅助方法。当然，这些新方法使用`Path`，但它们也可以与那些`java.io`包里基于流的类进行互操作。因此，你用一个方法就可以读取文件中的所有行或全部字节。

本节会向你介绍打开文件（带选项）的过程，以及一小组常用的文件读/写例子。让我们先从打开文件的不同方式开始。

1. 打开文件

Java 7可以直接用带缓冲区的读取器和写入器或输入输出流（为了和以前的Java I/O代码兼容）打开文件。下面的代码演示了Java 7如何用`Files.newBufferedReader`方法打开文件并按行读取其中的内容。

```
Path logFile = Paths.get("/tmp/app.log");
try (BufferedReader reader =
          Files.newBufferedReader(logFile, StandardCharsets.UTF_8)) {
  String line;
  while ((line = reader.readLine()) != null) {
     ...
  }
}
```

打开一个用于写入的文件也很简单。

```
Path logFile = Paths.get("/tmp/app.log");

try (BufferedWriter writer =
     Files.newBufferedWrite(logFile, StandardCharsets.UTF_8,
                            StandardOpenOption.WRITE)) {
  writer.write("Hello World!");
 ..
}
```

注意StandardOpenOption.WRITE选项的使用，这是可以添加的几个OpenOption变参之一。它可以确保写入的文件有正确的访问许可。其他常用的文件打开选项还有READ和APPEND。

与InputStream和OutputStream的交互是通过Files.newInputStream(Path,OpenOption...)和Files.newOutputStream(Path,OpenOption...)实现的。它们为过去基于java.io包的I/O和新的基于java.nio包的文件I/O之间架起了一座桥梁。

> **提示** 在处理String时，不要忘了查看它的字符编码。忘记设置字符编码（通过Standard-Charsets类，比如new String(byte[],StandardCharsets.UTF_8)）可能导致不可预料的字符编码问题。

前面的代码片段还是用Java 6及之前版本编写的读取和写入文件代码，仍然属于比较繁琐的底层代码。而Java 7具备更高层的抽象能力，可以帮你避免很多不必要的繁琐编码工作。

2. 简化读取和写入

辅助类Files有两个辅助方法，用于读取文件中的全部行和全部字节。也就是说你没必要再用while循环把数据从字节数组读到缓冲区里去。下面的代码演示了如何调用辅助方法。

```
Path logFile = Paths.get("/tmp/app.log");
List<String> lines = Files.readAllLines(logFile, StandardCharsets.UTF_8);
byte[] bytes = Files.readAllBytes(logFile);
```

对于某些软件来说，什么时候读、写是个问题，特别是在处理属性文件或日志时。这时就该文件修改通知系统大显身手了。

2.4.5 文件修改通知

在Java 7中可以用java.nio.file.WatchService类监测文件或目录的变化。该类用客户线程监视注册文件或目录的变化，并且在检测到变化时返回一个事件。这种事件通知对于安全监

测、属性文件中的数据刷新等很多用例都很有用。是现在某些应用程序中常用的轮询机制（相对而言性能较差）的理想替代品。

下面的代码用WatchService监测用户karianna主目录的变化，每当发现变化时就会在控制台中输出一个事件通知。和很多持续轮询的设计一样，它也需要一个轻量的退出机制。代码清单2-7如下所示：

代码清单2-7　使用WatchService

```
import static java.nio.file.StandardWatchEventKinds.*;

try
{
  WatchService watcher =
    FileSystems.getDefault().newWatchService();            ❶ 监测变化

  Path dir =
    FileSystems.getDefault().getPath("/usr/karianna");

  WatchKey key = dir.register(watcher, ENTRY_MODIFY);

  while(!shutdown)                                          ❷ 检查shutdown标志
  {
    key = watcher.take();                                   ❸ 得到下一个key
    for (WatchEvent<?> event: key.pollEvents())                 及其事件
    {
      if (event.kind() == ENTRY_MODIFY)
      {
        System.out.println("Home dir changed!");            ❹ 检查是否为
      }                                                         变化事件
    }
    key.reset();                                            ❺ 重置监测key
  }
}
catch (IOException | InterruptedException e)
{
  System.out.println(e.getMessage());
}
```

在得到默认的WatchService后，将karianna的主目录登记到变化监测名单中❶。然后在一个无限循环（直到shutdown标志改变）❷中执行WatcherService的take()方法，直到WatchKey的到来。一旦得到WatchKey，代码就遍历其WatchEvent进行检测❸。如果发现了类型为ENTRY_MODIFY的WatchEvent❹，就诏告天下karianna的主目录发生了变化！最后重置key❺准备迎接下一个事件，继续等待。

还有其他可以监测的事件，比如ENTRY_CREATE、ENTRY_DELETE和OVERFLOW（可以表明事件已经丢失或被丢弃了）。

接下来，我们要进入一个非常重要的、抽象的新API——用于数据的读写，使异步I/O成为现实的SeekableByteChannel。

2.4.6 SeekableByteChannel

Java 7引入SeekableByteChannel接口，是为了让开发人员能够改变字节通道的位置和大小。比如，应用服务器为了分析日志中的某个错误码，可以让多个线程去访问连接在一个大型日志文件上的字节通道。

JDK中有一个java.nio.channels.SeekableByteChannel接口的实现类——java.nio.channels.FileChannel。这个类可以在文件读取或写入时保持当前位置。比如说，你可能想要写一段代码读取日志文件中的最后1000个字符，或者向一个文本文件中的特定位置写入一些价格数据。

下面的代码展示了如何运用FileChannel的寻址能力读取日志文件中的最后1000个字符。

```
Path logFile = Paths.get("c:\\temp.log");
ByteBuffer buffer = ByteBuffer.allocate(1024);
FileChannel channel = FileChannel.open(logFile, StandardOpenOption.READ);
channel.read(buffer, channel.size() - 1000);
```

FileChannel类的寻址能力意味着开发人员可以更加灵活地处理文件内容。我们期待能由此产生一些有趣的开源项目，比如针对大型文件的并行访问。随着对该接口的不断扩展，可能还会有网络数据流的续传。

NIO.2 API中下一个主要修改是异步I/O，它可以使用多个后台线程读写文件、套接字和通道中的数据。

2.5 异步 I/O 操作

NIO.2另一个新特性是异步能力，这种能力对套接字和文件I/O都适用。异步I/O其实只是一种在读写操作结束前允许进行其他操作的I/O处理。实际上，就是可以充分利用最新的硬件和软件特性，比如多核CPU及操作系统对套接字和文件处理的支持。对于任何想在服务器端和系统级编程领域占有一席之地的编程语言来说，异步I/O都是必不可少的特性。我们相信，Java在服务器端编程语言中所取得的重要地位会因为该特性得以延续。

举个简单的例子，想象一下你要把100GB的数据写入文件系统或网络套接字中。如果你用的是老版本的Java，在同时把数据写入文件或套接字的多个区域时，必须亲自动手用java.util.concurrent写多线程代码。当然，同时进行多路读取也不容易。除非你写的代码十分巧妙，否则在使用I/O时也会阻塞主线程，这意味着在你完成漫长的I/O操作之前，除了等待，还是等待。

提示 如果你还没接触过NIO通道，也许可以趁此机会充实下你的知识结构。不过这个领域的新内容很少，但在继续本节内容之前，我们建议你去看看Ron Hitchens写的*Java NIO*（O'Reilly，2002）一书，你会从中获益匪浅。

Java 7中有三个新的异步通道：

- AsynchronousFileChannel——用于文件I/O；
- AsynchronousSocketChannel——用于套接字I/O，支持超时；
- AsynchronousServerSocketChannel——用于套接字接受异步连接。

使用新的异步I/O API时，主要有两种形式，将来式和回调式。有趣的是，这些异步API用到了第4章讨论的一些现代并发技术，所以这真是让你先睹为快了。

我们会从异步文件访问的将来式开始。希望你已经用过这种并发技术，但如果没有用过，也不用担心，本节将会讲解得非常详细，即便是刚接触这个话题的新手也能看明白。

2.5.1　将来式

NIO.2 API的设计人员用将来式（future）这个术语来表明使用java.util.concurrent.Future接口。当你希望由主控线程发起I/O操作并轮询等待结果时，一般都会用将来式异步处理。

将来式用现有的java.util.concurrent技术声明一个Future，用来保存异步操作的处理结果。这很关键，因为这意味着当前线程不会因为比较慢的I/O操作而停滞。相反，有一个单独的线程发起I/O操作，并在操作完成时返回结果。与此同时，主线程可以继续执行其他需要完成的任务。在其他任务结束后，如果I/O操作还没有完成，主线程会一直等待。图2-3演示了一个用将来式读取大型文件的过程。（代码清单2-8是相应的实现代码。）

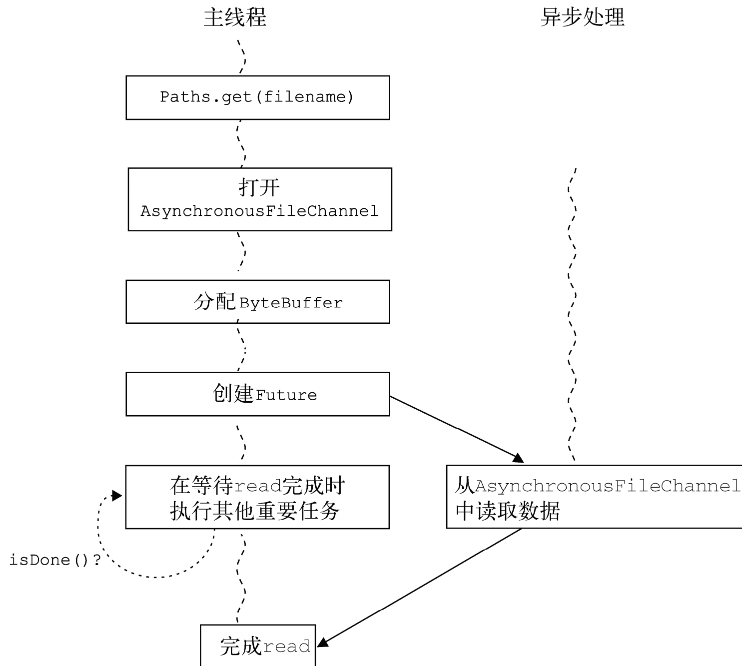


图2-3　将来式异步读取

通常会用`Future get()`方法（带或不带超时参数）在异步I/O操作完成时获取其结果。假设你要从硬盘上的文件里读取100 000个字节，在旧版的Java中，你需要等待数据读取完成（除非你实现了一个线程池，而且工作线程使用`java.util.concurrent`技术，这可不是件轻松的事儿）。而在Java 7中，主线程可以在读取数据的同时继续完成其他工作，如下面的代码所示。

代码清单2-8　异步I/O——将来式

```
try
{
  Path file = Paths.get("/usr/karianna/foobar.txt");

  AsynchronousFileChannel channel =
  AsynchronousFileChannel.open(file);

  ByteBuffer buffer = ByteBuffer.allocate(100_000);
  Future<Integer> result = channel.read(buffer, 0);

  while(!result.isDone())
  {
    ProfitCalculator.calculateTax();
  }

  Integer bytesRead = result.get();
  System.out.println("Bytes read [" + bytesRead + "]");
}
catch (IOException | ExecutionException | InterruptedException e)
{
  System.out.println(e.getMessage());
}
```

❶ 异步打开文件
❷ 读取100 000字节
❸ 干点儿别的事情
❹ 获取结果

上面的代码一开始先用后台进程中打开一个`AsynchronousFileChannel`读/写foobar.txt❶。接下来的这一步是为了让I/O处理能跟发起它的线程同步进行。因为采用`AsynchronousFileChannel`，并用`Future`保存读取结果，所以会自动采用并发的I/O处理❷。在读取数据时，主线程可以继续执行任务（比如算一下要交多少税）❸。最后，当任务完成时，你可以检查数据读取结果❹。

一定要注意，我们在这里用`isDone()`手工判定result是否结束。通常情况下，result或结束（主线程会继续执行），或等待后台I/O完成。

你可能会好奇这究竟是怎么实现的。长话短说，API/JVM为执行这个任务创建了线程池和通道组。另外，你也可以自己提供和配置一个。解释其中的细节颇费口舌，并且官方文档都解释过了，所以我们只是直接引用了`AsynchronousFileChannel`的Javadoc：

> `AsynchronousFileChannel`会关联线程池，它的任务是接收I/O处理事件，并分发给负责处理通道中I/O操作结果的结果处理器。跟通道中发起的I/O操作关联的结果处理器确保是由线程池中的某个线程产生的。

如果在创建`AsynchronousFileChannel`时没有为其指明线程池，那就会为其分配一个系统默认的线程池（可能会和其他通道共享）。默认线程池是由`AsynchronousChannelGroup`类定义的系统属性进行配置的。

此外还有一种被称为回调的技术。有些开发人员可能会发现回调式用起来更方便，因为它很像Swing、消息和其他Java API中出现过的事件处理技术。

2.5.2　回调式

与将来式相反，回调式（callback）所采用的事件处理技术类似于在Swing UI编程时采用的机制。其基本思想是主线程会派一个侦查员`CompletionHandler`到独立的线程中执行I/O操作。这个侦查员将带着I/O操作的结果返回到主线程中，这个结果会触发它自己的`completed`或`failed`方法（你会重写这两个方法）。

在异步事件刚一成功或失败并需要马上采取行动时，一般会用回调式。比如在读取对盈利计算业务处理至关重要的金融数据时，如果读取失败了，你最好马上就执行回滚操作，或进行异常处理。

在异步I/O活动结束后，接口`java.nio.channels.CompletionHandler<V,A>`会被调用，其中V是结果类型，A是提供结果的附着对象。此时必须已经有了该接口的`completed(V,A)`和`failed(V,A)`方法的实现，你的程序才能知道在异步I/O操作成功完成或因某些原因失败时该如何处理。图2-4展示了这一过程（代码清单2-9是该过程的实现代码）。

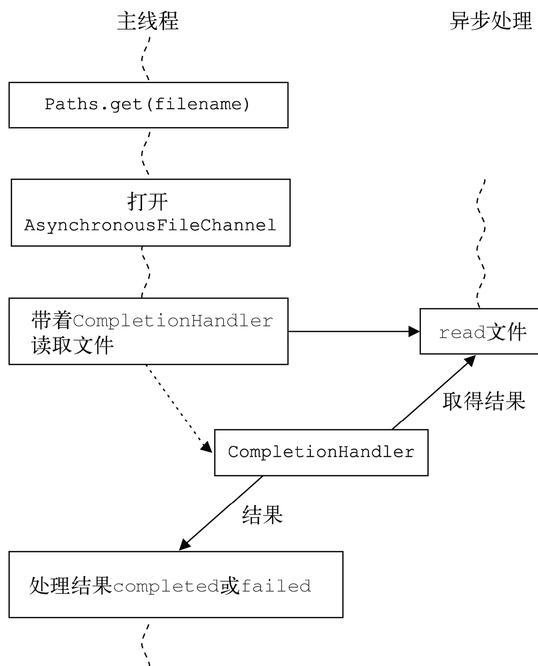


图2-4　回调式异步读取

在下例中，你又一次从foobar.txt文件中读取了100 000字节的数据，用`CompletionHandler<Integer,ByteBuffer>`声明是成功或是失败。

代码清单2-9 异步 I/O——回调式

```
try
{
  Path file = Paths.get("/usr/karianna/foobar.txt");
  AsynchronousFileChannel channel =
    AsynchronousFileChannel.open(file);               以异步方式打开文件

  ByteBuffer buffer = ByteBuffer.allocate(100_000);

  channel.read(buffer, 0, buffer,
    new CompletionHandler<Integer, ByteBuffer>()        从通道中读取数据
  {
    public void completed(Integer result,
                          ByteBuffer attachment)
    {
      System.out.println("Bytes read [" + result + "]");   读取完成时的
    }                                                       回调方法

    public void failed(Throwable exception, ByteBuffer attachment)
    {
      System.out.println(exception.getMessage());
    }
  });
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```

本节中的两个例子都是基于文件的，但将来式和回调式异步访问也适用于 AsynchronousServerSocketChannel和AsynchronousSocketChannel。开发人员可以用它们编写程序来处理网络套接字，比如语音IP或写出性能更优异的客户端和服务器端软件。

接下来的一系列变化统一了套接字和通道，让你可以将套接字和通道交互的管理归结到API中。

2.6 Socket 和 Channel 的整合

应用软件对网络接入的需求比以往任何时候都要迫切。仿佛一夜之间，家里所有东西都要联网了。在旧版Java中，套接字和通道结合得并不是很好，将它们两个配合在一起是件棘手的事情。因此Java 7推出了NetworkChannel，把Socket和Channel结合到一起，让开发人员可以轻松应对。

编写底层网络代码算是专业领域。如果你的工作领域与此无关，完全可以跳过这一节！但如果恰好你就是干这个的，你可以在本节对Java 7的新特性有一个初步了解。

我们先来看看套接字和通道在Javadoc中的定义，重温一下它们在Java中扮演的角色：

java.nio.channels包

定义通道，表示连接到执行I/O操作的实体，比如文件和套接字。定义用于多路传输、非阻塞I/O操作的选择器。

java.net.Socket类

该类实现了客户端套接字（也称为“套接字”）。套接字是两个机器间通信的端点。

在旧版Java中，为了执行I/O操作，比如向TCP端口中写入数据，你需要将通道绑定到Socket的实现类上，但Channel和Socket彼此之间却有“代沟”：

- 在旧版Java中，为了配置套接字选项和绑定在套接字上，必须把通道和套接字的API整合在一起；
- 在旧版Java中，不能利用平台特定的套接字行为。

让我们来看看新接口NetworkChannel和其子接口MulticastChannel对这两个领域做的“整理”工作。

2.6.1 NetworkChannel

新接口java.nio.channels.NetworkChannel代表一个连接到网络套接字通道的映射。它定义了一组实用的方法，比如查看及设置通道上可用的套接字选项等。下面的代码运用这些方法输出互联网套接字地址在端口3080上所支持的选项，设置IP服务条款选项以及确认套接字通道上的SO_KEEPALIVE选项。

代码清单2-10 NetworkChannel选项

```
SelectorProvider provider = SelectorProvider.provider();
try
{
  NetworkChannel socketChannel =
                           provider.openSocketChannel();
  SocketAddress address = new InetSocketAddress(3080);
  socketChannel = socketChannel.bind(address);

  Set<SocketOption<?>> socketOptions =
    socketChannel.supportedOptions();
  System.out.println(socketOptions.toString());

  socketChannel.setOption(StandardSocketOptions.IP_TOS,
                          3);
  Boolean keepAlive =
     socketChannel.getOption(StandardSocketOptions.SO_KEEPALIVE);
  ..
  ..
}
catch (IOException e)
{
    System.out.println(e.getMessage());
}
```

将**NetworkChannel**绑定到端口3080上

检查套接字选项

设置套接字的**ToS**（服务条款）选项

获取**SO_KEEPALIVE**选项

此外，NetworkChannel的出现使得多播操作成为可能。

2.6.2 MulticastChannel

像BitTorrent这样的对等网络程序一般都具备多播的功能。在Java的早期版本中，虽然拼凑一下也能实现多播，但却没有很好的API抽象层。Java 7中的新接口MulticastChannel解决了这个问题。

术语多播（或组播）表示一对多的网络通讯，通常用来指代IP多播。其基本前提是将一个包发送到一个组播地址，然后网络对该包进行复制，分发给所有接收端（注册到组播地址中），如图2-5所示。

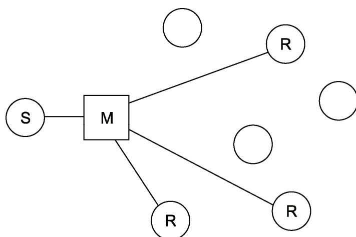


图2-5　多播示例

为了让新来的NetworkChannel加入多播组，Java 7提供了一个新接口java.nio.channels.MulticastChannel及其默认实现类DatagramChannel。也就是说你可以很轻松地对多播组发送和接收数据。

下面的代码说明了如何加入IP地址为180.90.4.12的多播组,并对其发送和接收系统状态信息。

代码清单2-11　NetworkChannel选项

```
try
{
  NetworkInterface networkInterface =
  NetworkInterface.getByName("net1");                  选择网络接口

  DatagramChannel dc =
  DatagramChannel.open(StandardProtocolFamily.INET);   打开DatagramChannel

  dc.setOption(StandardSocketOptions.SO_REUSEADDR,
              true);
  dc.bind(new InetSocketAddress(8080));
  dc.setOption(StandardSocketOptions.IP_MULTICAST_IF,
               networkInterface);                      将通道设置为多播
  InetAddress group =
              InetAddress.getByName("180.90.4.12");
  MembershipKey key = dc.join(group, networkInterface); 加入多播组
}
catch (IOException e)
{
  System.out.println(e.getMessage());
}
```

到此为止，我们对NIO.2 API的初步研究已经结束了。希望你喜欢这次行色匆匆的旅程!

2.7　小结

硬件和软件I/O的发展突飞猛进，而Java 7也紧随其后，充分利用了NIO.2的新API。Java 7提供的新类库可以用来处理位置（Path），用来在文件系统上执行操作，比如处理文件、目录、符

号链接等。特别值得一提的是，在平台特性的支持下，Java 7可以任意穿梭于文件系统中，并能够处理大型目录结构。

NIO.2致力于为那些通常需要大量编码工作的任务提供一站式的解决办法。尤其是新的`File`工具类，它有很多辅助方法，比起原来的`java.io.File`，它使得编写文件I/O代码更快，也更简单。

异步I/O是一个强大的新特性，可以保证在处理大文件时性能不受到显著影响。它对网络套接字和通道流量异常繁忙的程序也很有帮助。

NIO.2也用到了来自Coin项目（第1章）的新特性。这使得在Java 7中处理I/O比以往的版本更安全，而且所需的代码会更少。

现在是时候进入本书的第二部分了。让你的大脑准备好迎接挑战吧！依赖注入（Dependency Injection）、现代并发（Modern Concurrency）和基于Java的软件系统性能调优，这些都等着你去探索，把你喜爱的公爵（Duke）[①]杯加满咖啡，准备好向前冲吧！

① Duke是Java的吉祥物！http://kenai.com/projects/duke/pages/Home。

Part 2 第二部分

关键技术

本书的这一部分（第 3 章 ~ 第 6 章）全部是对 Java 中的关键编程知识和技术的探索。

本部分的开篇之章是关于依赖注入的，这是一项对代码解耦并增强其可测试性和易读性的通用技术。除了依赖注入的基础知识，我们还介绍了它的演进过程，并探讨了一个最佳实践是如何变成设计模式并形成一个框架的（最终甚至变成了 Java 标准）。

之后，我们会探究出现在硬件领域的多核 CPU 革命。优秀的 Java 开发人员要了解 Java 的并发能力，并知道如何利用它们充分发挥现代处理器的效用。尽管 Java 自 2006 年（Java 5）就大力支持并发编程，但人们对这一领域的理解和应用仍然很少，所以我们会用一整章的内容介绍它。

你将看到 Java 内存模型，以及这个模型中的线程和并发是如何实现的。一旦你掌握了这些理论知识，我们就会指导你用 `java.util.concurrent` 包及其他一些特性为 Java 并发实战打下基础。

接下来，我们会介绍类加载。很多 Java 开发人员不太明白 JVM 如何加载、链接和验证类。所以当某些类的“错误”版本由于某种类加载冲突被执行时，他们会备感沮丧并浪费很多时间。

我们还会谈到 Java 7 的 `MethodHandle`、`MethodType` 和动态调用，让用 Reflection（反射）编码的开发人员能以一种更快、更安全的方式完成相同的任务。

能够深入到 Java 类文件的内部和它所包含的字节码中是非常强的调试技能。我们会向你展示如何用 `javap` 浏览和理解字节码的含义。

性能调优经常被当做一门艺术，而不是科学。跟踪和解决性能问题经常会占用开发团队大量的时间和精力。在第 6 章，也就是本部分的最后一章，我们会教你评测（而不是猜测），并且告诉你“传说中的调优”是错误的。我们会给你指出一条直指性能问题核心的科学之路。

我们特别关注垃圾回收（GC）和即时（JIT）编译器，这是 JVM 中能够影响性能的两个主要部分。除了其他与性能有关的知识，你还将学到如何阅读 GC 日志，以及如何用免费的 Java VisualVM（jvisualvm）工具分析内存的使用情况。

读完第二部分之后，你就不再是个只想着 IDE 中那些源码的开发人员了。你将知道 Java 和 JVM 的内部工作机制，并能够充分发挥这个星球上最强大的通用 VM（这么说并不为过）的潜力。

第 3 章

依赖注入

3

本章内容

- 控制反转（IoC）和依赖注入（DI）
- 掌握依赖注入技术为什么如此重要
- JSR-330如何统一了Java中的DI
- 常见的JSR-330注解，比如`@Inject`
- Guice 3简介，JSR-330的参考实现（RI）

大约从2004年开始，依赖注入（控制反转的一种形式）就是Java开发主流中一个重要的编程范式[①]。简言之，使用DI技术可以让对象从别处得到依赖项，而不是由它自己来构造。使用DI有很多好处，它能降低代码之间的耦合度，让代码更易于测试、更易读。

本章会先对DI理论以及其给代码带来的好处进行强化。即便你用过IoC/DI框架，本章内容亦能帮你更深入地了解DI的本质。如果你刚刚开始接触DI框架（许多人都是如此），那本章中的内容对你就尤为重要了。

你将会了解Java DI的官方标准JSR-330，并从中了解到Java DI标准注解集的幕后故事。随后，我们会介绍JSR-330的参考实现（RI）Guice 3——一个众所周知的轻量、精巧的DI框架。

我们先来看一些理论知识，好让你明白这个范式大行其道的原因，以及你为什么需要掌握它。

3.1　知识注入：理解 IoC 和 DI

为什么需要了解控制反转（IoC、依赖注入（DI）以及它们的基本原理？对于这个问题，仁者见仁智者见智。如果你在知名的问答网站programmers.stackexchange.com上问这个问题，肯定会得到很多不同的答案！

你可能只是刚开始使用不同的DI框架并学习网上的示例，但如果你能够掌握对象关系映射（Object Relational Mapping，ORM）框架，比如Hibernate，你就可以变成编程高手。

① 范式（paradigm）在1960年之后是指在科学领域和知识论行文中的思维方式。——译者注

本节首先介绍核心术语IoC和DI背后的一些原理，并探讨使用这一范式的好处。为了让这些概念不至于那么抽象，我们会以`HollywoodService`为例展示它的转变过程——从自己构造依赖项变成被注入依赖项。

我们先从IoC开始，这个术语经常被（错误地）和DI互换使用。[①]

3.1.1 控制反转

在使用非IoC范式编程时，程序逻辑的流程通常是由一个功能中心来控制的。如果设计得好，这个功能中心会调用各个可重用对象中的方法执行特定的功能。

使用IoC，这个“中心控制”的设计原则会被反转过来。调用者的代码处理程序的执行顺序，而程序逻辑则被封装在接受调用的子流程中。

IoC也被称为好莱坞原则，其思想可以归结为会有另一段代码拥有最初的控制线程，并且由它来调用你的代码，而不是由你的代码调用它。

好莱坞原则——“不要给我们打电话，我们会打给你”

好莱坞经纪人总是给人打电话，而不让别人打给他们！如果你曾经跟好莱坞经纪人提议，在明年夏天筹划一个“让Java程序员成为拯救世界的英雄”的大片，你也许会深谙其道。

换一种方式来看IoC，回想一下视频游戏Zork(http://en.wikipedia.org/wiki/Zork)用户界面的发展过程，从早期由命令行中的文本控制到如今用图形用户界面控制。

在命令行版本中，用户界面只有一个空白提示符，等着用户输入。当用户输入“向东”或者“Grue，快逃”的指令后，游戏的主应用逻辑会调用恰当的事件处理器来处理这些指令，并返回结果。这里的关键点是应用逻辑要控制调用哪个事件处理器。

而在GUI版本中，IoC开始发挥作用。由GUI框架来控制调用事件处理器，而不是由应用逻辑。当用户点击了一个动作，比如“向东”时，GUI框架会直接调用对应的事件处理器，而应用逻辑可以把重点放在处理动作上。

程序的主控被反转了，将控制权从应用逻辑中转移到GUI框架。[②]

IoC有几种不同的实现，包括工厂模式、服务定位器模式，当然，还有依赖注入。这一术语最初由Martin Fowler在“控制反转容器和依赖注入模式”[③]中提出，然后迅速传遍大街小巷，反响强烈。

---

① 从字面上来看，IoC是指一种机制，使用这种机制的用例很多，实现方式也很多。DI只是其中一种具体用例的具体实现方式。但因为DI非常流行，所以人们经常误以为IoC就是DI，并且认为DI这种叫法比IoC更贴切。这是来自stackoverflow的更全面解释（英文）：http://stackoverflow.com/questions/6550700/inversion-of-control-vs-dependency-injection。——译者注

② 程序中出现了专门用来实现调用和控制逻辑的GUI框架，应用逻辑中的代码只需关注应用请求的处理。——译者注

③ 在Martin Fowler的网站http://martinfowler.com/中搜索Dependency Injection，你就可以找到这篇文章。

3.1.2　依赖注入

依赖注入是IoC的一种特定形态，是指寻找依赖项的过程不在当前执行代码的直接控制之下。虽然你也可以自己编写代码实现依赖注入机制，但大多数开发人员都会使用自带IoC容器的第三方DI框架，比如Guice。

注意　可以把IoC容器看做运行时环境。Java中为依赖注入提供的容器有Guice、Spring和PicoContainer。

IoC容器可以提供实用的服务，比如确保一个可重用的依赖项会被配置成单例模式。我们在3.3节介绍Guice时会探讨它的一些服务。

提示　把依赖项注入对象的方法有很多种。可以用专门的DI框架，但也可以不这么做！显式地创建对象实例（依赖项）并把它们传入对象中也可以和框架注入做的一样好。[①]

与很多编程范式一样，理解使用DI的原因很重要。我们在表3-1中总结了它的主要好处。

表3-1　DI的好处

好　处	描　述	例　子
松耦合	你的代码不再紧紧地绑定到依赖项的创建上了。如果能与面向接口编程的技术相结合，意味着你的代码再也不用紧紧地绑定到依赖项的具体实现上了	`HollywoodService`对象不再需要创建它所需的`SpreadsheetAgentFinder`对象，而是使用从外部传入的对象。如果面向接口编程，`HollywoodService`可以接受任何类型的`AgentFinder`传入
易测性	作为松耦合的延伸，还有个特殊的用例值得一提。为了测试，可以把测试替身[②]作为依赖项注入到对象中	你可以注入一个总是返回相同价格的虚设票价服务，而不是使用“真实”的价格服务，因为它是外部服务，而且有时无法访问
更强的内聚性	你的代码可以专注于执行自己的任务，不用为了载入和配置依赖项而分心。这样还能增强代码的可读性	你的DAO对象可以专注于查询工作，不用考虑JDBC驱动的细节
可重用组件	作为松耦合的延伸，你的代码应用范围会更加宽广，那些可以提供自己特定实现的用户都可以使用你的代码	一个积极进取的软件开发人员可能会卖给你一个LinkedIn代理人查找器
更轻盈的代码	你的代码不再需要跨层传递依赖项，而是可以在需要依赖项的地方直接注入	你不再需要把JDBC驱动的细节信息从service类往下传递，而是直接在真正需要它的DAO中直接注入这个驱动实例

把普通代码改写成依赖注入的代码是掌握这些理论的最佳方法，所以我们进入下一节吧。

---

① 感谢Thiago Arrais（http://stackoverflow.com/users/17801/thiago-arrais）提供了这个提示。

② 第11章会详细介绍测试替身。

3.1.3 转成DI

本节会重点介绍如何把不用IoC的代码变成使用工厂（或服务定位器）模式的代码，再变成使用DI的代码。在这些转变之后有一个共同的关键技术，即面向接口编程。使用面向接口编程，甚至可以在运行时更换对象。

注意 本节的目的是巩固你对DI的理解。因此某些比较套路化的代码被省略掉了。

假设你刚接手了一个小项目，要找出所有对Java开发人员比较友善的好莱坞经纪人。在下面的代码中，`AgentFinder`接口有两个实现类：`SpreadSheetAgentFinder`和`WebServiceAgentFinder`。

代码清单3-1 接口`AgentFinder`及其实现类

```
public interface AgentFinder
{
  public List<Agent> findAllAgents();
}

public class SpreadsheetAgentFinder implements AgentFinder
{
  @Override
  public List<Agent> findAllAgents(){ ... }      <-┐
}                                                   │
                                                    │ 很多实现细节
public class WebServiceAgentFinder implements AgentFinder
{                                                   │
  @Override                                         │
  public List<Agent> findAllAgents(){ ... }      <-┘
}
```

为了使用经纪人查找器，项目中有个默认的`HollywoodServic`类，它会从`Spreadsheet-AgentFinder`里得到一个经纪人列表，并根据是否友善对他们进行过滤，最终返回友善的经纪人列表。如下面的代码所示。

代码清单3-2 `HollywoodService`——自己创建`AgentFinder`的具体实现类实例

```
public class HollywoodService
{                                                  ❶ 使用Spreadsheet-
                                                     AgentFinder
  public static List<Agent> getFriendlyAgents()
  {
    AgentFinder finder = new SpreadsheetAgentFinder();   <-❶
    List<Agent> agents = finder.findAllAgents();       <- 调用接口方法
    List<Agent> friendlyAgents =
        filterAgents(agents, "Java Developers");
    return friendlyAgents;                          <- 返回友善的经纪人
  }
```

```
    public static List<Agent> filterAgents(List<Agent> agents,
        String agentType)
    {
      List<Agent> filteredAgents = new ArrayList<>();
      for (Agent agent:agents) {
        if (agent.getType().equals("Java Developers")) {
          filteredAgents.add(agent);
        }
      }
      return filteredAgents;
    }
  }
```

再看一遍上面代码里的HollywoodService，注意到了吗？它被SpreadsheetAgentFinder这个AgentFinder的具体实现死死地黏上了❶。

过去这种黏糊糊的实现对Java开发者来说司空见惯，不胜其烦！为了解决这些共性问题，设计模式应运而生。一开始，很多开发人员用工厂模式和服务定位器模式的各种变体来解决这类问题，它们全都是IoC的一种。

1. 使用工厂和/或服务定位器模式的HollywoodService

抽象工厂、工厂方法或服务定位器模式中的某个（或它们的某种组合）通常用来解决这种被依赖项黏上的问题。

注意　工厂方法和抽象工厂模式在Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides写的《设计模式：可复用面向对象软件的基础》（Addison-Wesley Professional，1994）中有所论述。服务定位器模式出现在Deepak Alur、John Crupi和Dan Malks编写的《J2EE核心模式》第二版（Prentice Hall，2003）中。

下面这个版本的HollywoodService类用AgentFinderFactory实现对AgentFinder的动态选择。

代码清单3-3　HollywoodService由工厂负责提供AgentFinder

```
public class HollywoodServiceWithFactory {

  public List<Agent>
         getFriendlyAgents(String agentFinderType)          ❶ 注入agentFinderType
  {
    AgentFinderFactory factory =
                 AgentFinderFactory.getInstance();          ❷ 通过工厂实例获取
    AgentFinder finder =                                       AgentFinder
                 factory.getAgentFinder(agentFinderType);
    List<Agent> agents = finder.findAllAgents();
    List<Agent> friendlyAgents =
        filterAgents(agents, "Java Developers");
    return friendlyAgents;
  }
```

```
  public static List<Agent> filterAgents(List<Agent> agents,
      String agentType)
  {
    ...                          <-- 与代码清单3-2中的实现一样
  }
}
```

如你所见，现在没有特定的AgentFinder实现类来黏你了。注入agentFinderType❶，让AgentFinderFactory根据这一类型挑选令人满意的AgentFinder供你享用❷。

这和DI相当接近了，但还有两个问题。

- 代码中注入的是一个引用凭据（agentFinderType），而不是真正实现AgentFinder的对象。
- 方法getFriendlyAgents中还有获取其依赖项的代码，达不到只关注自身职能的理想状态。

随着开发人员编写更清晰代码的意愿不断增强，DI技术也越来越流行，正在逐步取代工厂和服务定位器模式。

2. 使用DI的HollywoodService

你可能已经猜出接下来重构代码该做什么了！下一步要让AgentFinder直接提供所需的getFriendlyAgents方法。代码如下所示：

代码清单3-4　HollywoodService——纯手工DI注入AgentFinder

```
public class HollywoodServiceWithDI
{

  public static List<Agent>
              emailFriendlyAgents(AgentFinder finder)     <-- ❶ 注入AgentFinder
  {
    List<Agent> agents = finder.findAllAgents();          <-- ❷ 执行查找逻辑
    List<Agent> friendlyAgents =
        filterAgents(agents, "Java Developers");
    return friendlyAgents;
  }

  public static List<Agent> filterAgents(List<Agent> agents,
      String agentType)
  {
    ...                          <-- 参见代码清单3-2
  }
}
```

看看这个纯手工打造的DI方案，AgentFinder被直接注入到getFriendlyAgents方法中❶。现在这个getFriendlyAgents方法干净利落，只专注于纯业务逻辑❷。

不过这种手工打造DI的生产方式还是有让人头疼的地方。如何配置AgentFinder具体实现的问题并没有解决，原本AgentFinderFactory要做的工作还是要找个地方完成。

所以，能够真正让我们脱离苦海的只有自带IoC容器的DI框架。打个比方，DI框架就是把你的代码包起来的运行时环境，在你需要时为你注入依赖项。

DI框架的优势在于它可以随时随地为你的代码提供依赖项。因为框架中有IoC容器，在运行时，你的代码需要的所有依赖项都会在那里准备好。

如果HollywoodService类使用标准的JSR-330注解（可以使用任何与JSR-330兼容的框架），那么它会是什么样子？

3. 使用JSR-330 DI的HollywoodService

来看看这个例子的最终版本，用框架来执行DI操作。在这里，DI框架用标准的JSR-330@Inject注解将依赖项直接注入到getFriendlyAgents方法中，代码如下所示：

代码清单3-5　HollywoodService——用JSR-330 DI注入AgentFinder

```
public class HollywoodServiceJSR330
{
  @Inject public static List<Agent> emailFriendlyAgents(AgentFinder finder)   ← JSR-330注入 AgentFinder ❶
  {
    List<Agent> agents = finder.findAllAgents();          | 执行查找逻辑
    List<Agent> friendlyAgents =                          |
        filterAgents(agents, "Java Developers");          |
    return friendlyAgents;
  }

  public static List<Agent> filterAgents(List<Agent> agents,
      String agentType)
  {
    ...                                       ← 参见代码清单3-2
  }
}
```

现在AgentFinder的某个具体实现（比如WebServiceAgentFinder）类的实例是由支持JSR-330@Inject注释的DI框架在运行时注入的❶。

提示　尽管JSR-330注解可以在方法上注入依赖项，但通常只用于构造方法或set方法中。下一节会对这一规范做进一步探讨。

让我们结合代码清单3-5中的HollywoodServiceJSR330类再来重温一下依赖注入对我们的帮助。

- 松耦合——HollywoodService不再依赖于AgentFinder的具体类来完成工作。
- 可测性——为了测试HollywoodService类，你可以注入一个返回固定数量经纪人的基本Java类（比如POJOAgentFinder），这在测试驱动的开发中被称为存根类。这对于单元测试来说非常完美，因为你不再需要Web服务、电子表格或其他第三方实现之类的东西。
- 更强的内聚性——你的代码不用再和工厂打交道，不用四下里去抓依赖项，只需要执行业务逻辑。
- 可重用的组件——假如有个开发人员想用你的API，现在需要注入一个他们定制的AgentFinder实现类，就说JDBCAgentFinder吧，想象一下他轻松惬意的表情吧。

- 更轻盈的代码——`HollywoodServiceJSR330`中的代码量与最初的`HollywoodService`相比明显减少了很多。

使用DI正在逐步成为优秀Java开发人员的标准实践，几个流行的容器都提供了优异的DI能力。然而就在不久之前，DI框架领域还是群雄割据，它们风格迥异，各行其是，使用IoC容器的配置标准都自成体系。即便遵循类似配置风格的框架（比如XML或Java注解），还是存在什么是共通的注解和配置这个问题。

新的DI标准化方式（JSR-330）就是要解决这个问题。它对大多数Java DI框架的核心能力做了很好的汇总。因为有DI框架（比如Guice）的内部工作机制作为其坚实的基础，所以接下来我们会深入探讨这一标准化方式。

3.2 Java 中标准化的 DI

从2004年开始，有几个用于依赖注入的IoC容器得到了广泛的应用（仅以Spring、Guice和PicoContainer为例）。曾几何时，它们在DI的配置方式上仍然各自为政，这使开发人员很难在不同框架之间迁移。

这一问题直到2009年5月才出现转机，DI社区的两大巨头Bob Lee（Guice）和Rod Johnson（SpringSource）宣布要齐心协力，共同打造一组标准的接口注解[①]。他们紧接着提出了JSR-330（`javax.inject`）规范请求，倡导Java SE的DI标准化。DI社区的各路诸侯纷纷响应，全部表示支持。

Java EE中的DI标准化情况如何？

Java企业应用从JEE 6开始构建了自己的依赖注入体系（即CDI），由JSR-299（Java EE平台中的上下文及依赖注入）规范确定，你可在http://jcp.org/中搜索JSR-299了解其详细信息。简言之，JSR-299构建在JSR-330基础之上，旨在为企业应用提供标准化的配置。[②]

自从`javax.inject`出现在Java中（Java SE 5、6和7都支持）以来，代码中就可以使用标准的依赖注入了，也可以在不同的DI框架中进行迁移。比如，你原来在轻量级的Guice框架中运行的代码，为了利用其丰富的特性，也可以迁移到Spring框架中去。

警告 实际上，代码迁移并不容易。一旦你的代码用到了仅由特定DI框架支持的特性，就不太可能摆脱这一框架了。尽管`javax.inject`包提供了常用DI功能的子集，但是你可能需要使用更高级的DI特性。正如你想象的那样，对于哪些特性应该作为通用的标准也是众说纷纭，很难统一。虽然现状不尽如人意，但Java毕竟朝DI框架的标准化方向迈出了一步。

---

① Bob Lee，“Announcing @javax.inject.Inject”（2009-05-08），www.theserverside.com/news/thread.tss?thread_id=54499。

② JSR-299（Java Contexts and Dependency Injection）目前由Redhat的Gavin King（Hibernate的创建者）主导，因为它比较新，所以设计理念上解决了以前DI框架中的一些问题，而且也不是非得在Java EE容器上才能使用，在Servlet容器上也可以使用。其参考实现为weld，详情请参见官网：http://www.seamframework.org/Weld。——译者注

为了理解最新的DI框架如何应用新标准，我们需要对javax.inject包进行一番研究。记住，javax.inject包只是提供了一个接口和几个注解类型，这些都会被遵循JSR-330标准的各种DI框架实现。也就是说，除非你在创建与JSR-330兼容的IoC容器（如果如此，向你致敬），通常不用自己实现它们。

我为什么要知道这东西怎么工作？

优秀的Java开发人员不能满足于只作为类库和框架的使用者，还要明白其内部的基本工作原理。在DI领域，不理解其原理可能会面临各种难缠的问题，比如依赖项配置错误、依赖项诡异地超出作用域、依赖项在不该共享时被共享以及分步调试离奇宕机等。

javax.inject的文档对这个包的目的做出了精彩的解释，所以我们全盘照搬过来了：

javax.inject包[1]

这个包指明了获取对象的一种方式，与传统的构造方法、工厂模式和服务定位器模式（比如JNDI）等相比，这种方式的可重用性、可测试性和可维护性都得到了极大提升。这种方式称为依赖注入，对于大多数非小型应用程序都很有帮助。

javax.inject包里包括一个Provider<T>接口和5个注解类型（@Inject、@Qualifier、@Named、@Scope和@Singleton），后面几节会逐一对它们进行介绍。先从@Inject开始。

3.2.1　**@Inject**注解

@Inject注解可以出现在三种类成员之前，表示该成员需要注入依赖项。按运行时的处理顺序，这三种成员类型是：

(1) 构造器

(2) 方法

(3) 属性

在构造器上使用@Inject时，其参数在运行时由配置好的IoC容器提供。比如在下面的代码中，运行时调用MurmurMessage类的构造器时，IoC容器会注入其参数Header和Content对象。

```
@Inject public MurmurMessage(Header header, Content content)
{
  this.header = header;
  this.content = content;
}
```

规范中规定向构造器注入的参数数量为0或多个，所以在不含参数的构造器上使用@Inject也是合法的。

① http://atinject.googlecode.com/svn/trunk/javadoc/javax/inject/package-summary.html。

警告 因为JRE无法决定构造器注入的优先级，所以规范中规定类中只能有一个构造器带`@Inject`注解。

也可以用`@Inject`注解方法，与构造器一样，运行时可注入参数的数量也可以是0或多个。但使用参数注入的方法不能声明为抽象方法，也不能声明其自身的类型参数[①]。下面这段代码在set方法前使用`@Inject`，这是注入可选属性的常用技术。

```
@Inject public void setContent(Content content)
{
  this.content = content;
}
```

向方法中注入参数的技术对于服务类方法来说非常有用，其所需的资源可以作为参数注入，比如向查询数据的服务方法中注入数据访问对象（DAO）。

提示 向构造器中注入的通常是类中必需的依赖项，而对于非必需的依赖项，通常是在set方法上注入。比如已经给出了默认值的属性就是非必需的依赖项。这一最佳实践已经成了惯例。

也可以直接在属性上注入（只要它们不是`final`），虽然这样做简单直接，但你最好不要用。因为这样可能会让单元测试更加困难。直接注入的语法也非常简单。

```
public class MurmurMessenger
{
  @Inject private MurmurMessage murmurMessage;
  ...
}
```

你可以在Javadoc中了解更多关于`@Inject`注解的详细内容，可以在其中找到哪些类型的值可以注入以及如何处理依赖循环。

对于`@Inject`，现在你应该不再感到陌生了。接下来就该了解如何限定（进一步标识）那些注入到你的代码中的对象了。

3.2.2 `@Qualifier`注解

支持JSR-330规范的框架要用注解`@Qualifier`限定（标识）要注入的对象。比如在IoC容器中有两个类型相同的对象，当把它们注入到你的代码中时，肯定要把它们区别开。图3-1解释了这一概念。

① 即不能使用Oracle网站上的Java教程(http://download.oracle.com/javase/tutorial/extra/generics/methods.html)中所讲的泛型方法技巧。

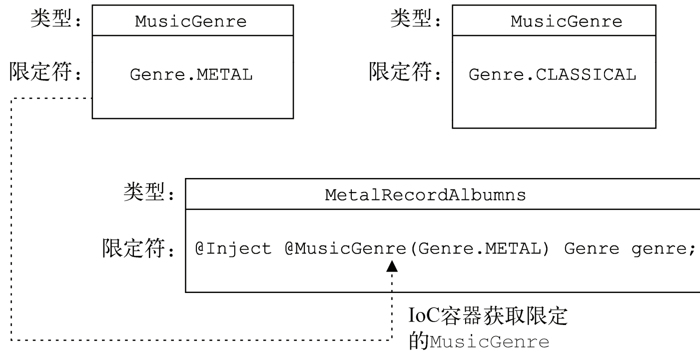


图3-1　用@Qualifier注解区分同一类型MusicGenre的不同bean

如果你用过由框架实现的限定符，应该知道要创建一个@Qualifier实现必须遵循如下规则。

- 必须标记为@Qualifier和@Retention(RUNTIME)，以确保该限定注解在运行时一直有效。
- 通常还应该加上@Documented注解，这样该实现就能加到API的公共Javadoc中了。
- 可以有属性。
- @Target注解可以限定其使用范围；比如将其使用范围限制为属性，而不是限定为属性的默认值和方法中的参数。

为了让你对上面的规则有直观的感受，下面给出一个@Qualifier实现。某音乐库框架中的IoC容器提供了一个限定符@MusicGenre，开发人员在创建MetalRecordAlbumns类时可以使用该限定符，以确保注入了正确的Genre。

```
@Documented
@Retention(RUNTIME)
@Qualifier
public @interface MusicGenre
{
  Genre genre() default Genre.TRANCE;
  public enum GENRE { CLASSICAL, METAL, ROCK, TRANCE }
}

public class MetalRecordAlbumns
{
  @Inject @MusicGenre(GENRE.METAL) Genre genre;
}
```

Java开发人员一般不需要自己创建@Qualifier注解，但要对各种IoC容器如何实现限定有个基本的了解。

JSR-330规范中要求所有IoC容器都要提供一个默认的@Qualifier注解：@Named。

3.2.3 @Named注解

@Named是一个特别的@Qualifier注解，借助@Named可以用名字标明要注入的对象。将@Named和@Inject一起使用，符合指定名称并且类型正确的对象会被注入。

在下面这个例子中，注入了名称为“murmur”和“broadcast”的两个MurmurMessage对象。

```
public class MurmurMessenger
{
  @Inject @Named("murmur") private MurmurMessage murmurMessage;
  @Inject @Named("broadcast") private MurmurMessage broadcastMessage;
  ...
}
```

尽管还有其他比较常用的限定注解，但最终只有@Named被选作JSR-330的标准限定注解，所有DI框架都要实现。

发起规范的各方支持者还在另外一个问题上达成了一致——同意用标准化接口来确定注入对象的生命周期。

3.2.4 @Scope注解

@Scope注解用于定义注入器（即IoC容器）对注入对象的重用方式。JSR-330规范中明确了如下几种默认行为。

- 如果没有声明任何@Scope注解接口的实现，注入器应该创建注入对象并且仅使用该对象一次。
- 如果声明了@Scope注解接口的实现,那么注入对象的生命周期由所声明的@Scope注解实现决定。
- 如果注入对象在@Scope实现中要由多个线程使用，则需要保证注入对象的线程安全性。关于线程及线程安全的更多细节，请参阅第4章。
- 如果某个类上声明了多个@Scope注解，或声明了不受支持的@Scope注解，IoC容器应该抛出异常。

DI框架管理注入对象的生命周期时不会超出这些默认行为划定的界限。有些IoC容器支持自己特有的@Scope，尤其是在Web前端领域，至少在JSR-299被广泛应用之前是这样。因为大家公认的通用@Scope实现只有@Singleton一个，所以JSR-330规范中仅确定了它这么一个标准的生命周期注解。

3.2.5 @Singleton注解

@Singleton注解接口在DI框架中应用广泛。在需要注入一个不会改变的对象时，就要用@Singleton。

单例模式

单例设计模式是为了确保类仅被实例化一次，详情参见由Erich Gamma, Richard Helm, Ralph Johnson, 和John Vlissides合著的《设计模式：可复用面向对象软件的基础》（Addison-Wesley Professional, 1994）第127页。请谨慎使用单例模式，因为它有时候会变成反模式。

大多数DI框架都将@Singleton作为注入对象的默认生命周期，无需显式声明。也就是说如果没有明确指定注入对象的生命周期，框架就会认为你想注入一个单例对象。如果你想显式声明它为单例对象，可以用下面这种方式：

```
public MurmurMessage
{
  @Inject @Singleton MessageHeader defaultHeader;
}
```

在这个例子中，我们假定defaultHeader从来不会改变（切实有效的静态数值），所以它可以作为单例对象注入。

最后，我们来讨论一下当你觉得标准注解无法满足你的需求时该怎么办。

3.2.6　接口Provider<T>

如果你想对由DI框架注入代码中的对象拥有更多的控制权，可以要求DI框架将Provider<T>接口实现注入对象[①](T)。控制对象的好处在于：

- 可以获取该对象的多个实例。
- 可以延迟获取该对象（延迟加载）。
- 可以打破循环依赖。
- 可以定义作用域，能在比整个被加载的应用小的作用域中查找对象。

该接口仅有一个T get()方法，这个方法会返回一个构造好的注入对象(T)。例如，在MurmurMessage需要依赖项Message对象时，可以向其构造方法中注入对应的Provider<T>接口实现的实例（Provider<Message>）。根据限定条件的不同，得到的Message对象也会不同。请看下面的代码。

代码清单3-6　使用接口Provider<T>

```
import com.google.inject.Inject;
import com.google.inject.Provider;

class MurmurMessage
{
  @Inject MurmurMessage (Provider<Message> messageProvider)
  {
    Message msg1 = messageProvider.get();          <-- 得到一个Message
```

① 原文如此，应为该类，后面还有几处笔误，请注意。——译者注

```
        if (someGlobalCondition)
        {
          Message copyOfMsg1 = messageProvider.get();
        }
        ...
      }
    }
```

❶ 得到Message的复本

注意上面的代码中是如何从Provider<Message>中获取第二个Message对象的。如果直接注入Message，就无法获取另外一个Message实例。在这个例子中，第二个注入对象仅在需要时才会加载❶。

我们已经对新javax.inject包背后的理论做了介绍，还给出了一些例子进行讲解。现在就该挽起袖子，用成熟的DI框架Guice实际操练一把了。

3.3　Java 中的 DI 参考实现：Guice 3

Guice（读作“Juice”）是由Bob Lee在大约2006年发起的开源项目，项目站点地址为http://code.google.com/p/google-guice/。你可以在网站上看到该项目的设计初衷、相关文档并下载运行本节示例代码所需的二进制JAR文件。

在Guice这样的DI框架里，你可以配置依赖项、绑定依赖项，并在使用@Inject注解（和它在JSR-330中的朋友们）注入依赖项时声明它们的作用域。

Guice 3是JSR-330规范的完整参考实现，本节所有内容都是基于Guice 3的。虽然Guice不仅仅是一个简单的DI框架，但我们关注的主要还是它的DI能力和示例代码，其中你可以使用JSR-330标准注解和Guice一起编写DI代码。

3.3.1　Guice新手指南

现在你已经了解JSR-330的各种注解了。可以借助Guice构建一个Java注入对象集合（包括它们的依赖项）。用Guice的话说，为了让注入器创建对象关系图，需要创建声明各种绑定关系的模块，其中绑定是用来明确要注入的具体实现类的。晕了没？不用担心，看到代码你就明白了，这些概念实际上非常简单。

提示　对象关系图、绑定、模块和注入器都是Guice里的常用语，如果你想用好Guice，最好尽快搞清楚它们是什么意思。

本节我们还是用HollywoodService的例子。一开始先创建一个拥有各种绑定关系的配置类（模块）。实际上这是Guice框架要管理的那些依赖项的外部配置。

在哪里下载Guice？

最新版的Guice 3可以从http://code.google.com/p/google-guice/downloads/list上下载，对应的文档可以在http://code.google.com/p/google-guice/wiki/Motivation?tm=6上找到。

要得到完整的IoC容器和DI支持，还需要下载Guice zip文件并将它解压到你选定的位置。为了在Java代码中使用Guice，要确保这些JAR文件包含在CLASSPATH中。

对于本书中后续代码示例而言，构建Maven时也会自动下载Guice 3的JAR文件。

我们先来创建一个定义绑定关系的`AgentFinderModule`。这个`AgentFinderModule`类扩展了`AbstractModule`，绑定关系在重写的`configure()`方法中声明。在本例中，当客户类`HollywoodService`要求`@Inject`一个`AgentFinder`的时候，就会绑定`WebServiceAgentFinder`类作为注入对象。我们在这里遵循构造方法注入的惯例，具体实现请见代码：

代码清单3-7　`HollywoodService`——用 Guice 注入`AgentFinder`

```
import com.google.inject.AbstractModule;

public class AgentFinderModule extends AbstractModule      <-- 扩展AbstractModule
{
  @Override                                                <-- 重写configure()方法
  protected void configure()
  {
    bind(AgentFinder.class).
               to(WebServiceAgentFinder.class);           <-- ❶ 绑定要注入的实现类
  }
}

public class HollywoodServiceGuice
{
  private AgentFinder finder = null;

  @Inject
  public HollywoodServiceGuice(AgentFinder finder)
  {
    this.finder = finder;
  }

  public List<Agent> getFriendlyAgents()
  {
    List<Agent> agents = finder.findAllAgents();
    List<Agent> friendlyAgents = filterAgents(agents, "Java Developers");
    return friendlyAgents;
  }

  public List<Agent> filterAgents(List<Agent> agents, String agentType)
  {
    ...                                                    <-- 同代码清单3-2
  }

}
```

绑定关系的确立在调用Guice的bind方法时发生，把要绑定的类（AgentFinder）传给它，然后调用to方法指明要注入到哪个实现类❶。

现在已经在模块中声明了绑定关系，可以让注入器构建对象关系图了。接下来我们要看看在独立Java程序和Web应用程序这两种情况下分别要如何实现。

1. 构建Guice对象关系图——独立Java程序

在标准的Java程序中，可以通过public static void main(String[] args)方法构建对象关系图。代码清单3-8如下所示。

代码清单3-8　HollywoodServiceClient——用Guice构建对象关系图

```
import com.google.inject.Guice;
import com.google.inject.Injector;
import java.util.List;

public class HollywoodServiceClient
{
  public static void main(String[] args)
  {
    Injector injector =
        Guice.createInjector(new AgentFinderModule());

    HollywoodServiceGuice hollywoodService =
        injector.getInstance(HollywoodServiceGuice.class);
   List<Agent> agents = hollywoodService.getFriendlyAgents();
    ...
  }
}
```

对于Web应用，情况稍有不同。

2. 构建Guice对象关系图——Web应用程序

在Web应用程序中，需要把guice-servlet.jar加到Web应用的类库中，然后在web.xml中添加下面的配置项：

```
<filter>
  <filter-name>guiceFilter</filter-name>
  <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>guiceFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

然后是标准动作，扩展ServletContextListener以便使用Guice的ServletModule（与代码清单3-7中的AbstractModule类似）。

```
public class MyGuiceServletConfig extends GuiceServletContextListener {
  @Override
  protected Injector getInjector() {
    return Guice.createInjector(new ServletModule());
  }
}
```

最后一步，把下面这些配置加到web.xml文件中，以便servlet容器在部署应用时触发该类。

```
<listener>
  <listener-class>com.java7developer.MyGuiceServletConfig</listener-class>
</listener>
```

经由注入器创建`HollywoodServiceGuice`，你得到了一个配置完备的类，马上就可以调用其中的`getFriendlyAgents`方法。

非常简单，对不对？没错，但这种把`WebServiceAgentFinder`绑定到`AgentFinder`上的情况很简单，而你所需要的绑定方式可能要比这个复杂，所以我们还需要了解一下如何定义更复杂的绑定方式。

3.3.2　水手绳结：Guice的各种绑定

Guice提供了多种绑定方式，官方文档列出的绑定类型如下所示：

- 链接绑定
- 绑定注解
- 实例绑定
- `@Provides`方法
- Provider绑定
- 无目标绑定
- 内置绑定
- 即时绑定

我们无意在此重复Guice的官方文档，因此只挑最常用的讲解一下，其中包括链接绑定、绑定注解和`@Provides`方法及`Provider<T>`绑定。

1. 链接绑定

链接绑定是最简单的绑定方式，代码清单3-6中配置`AgentFinderModule`时用的就是这种方式。这种绑定方式只是告诉注入器运行时应该注入实现类或扩展类（是的，可以直接注入子类）。

```
@Override
protected void configure()
{
  bind(AgentFinder.class).to(WebServiceAgentFinder.class);
}
```

你已经见过这种绑定代码了，让我们看看另外一种最常用的绑定方式：绑定注解。

2. 绑定注解

绑定注解是指将注入类的类型和额外的标识符组合起来，以标识恰当的注入对象。你可以自定义绑定注解（参见Guice在线文档），不过我们还是先介绍一下JSR-330标准注解`@Named`的用法。

在下面的例子中，你所熟悉的`@Inject`依然会出现，但它这次会与`@Named`注解联袂登场，以注入特定名称的`AgentFinder`。为了配置`@Named`绑定，需要在`AgentModule`中调用`annotatedWith`方法，代码清单3-9如下所示：

代码清单3-9 HollywoodService——使用@Named

```
public class HollywoodService
{
  private AgentFinder finder = null;

  @Inject
  public HollywoodService(@Named("primary") AgentFinder finder)    <- 使用@Named注解
  {
    this.finder = finder;
  }
}

public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure()
  {
    bind(AgentFinder.class)
      .annotatedWith(Names.named("primary"))    <- 与命名参数绑定在一起
      .to(WebServiceAgentFinder.class);
  }
}
```

现在你已经知道如何配置命名依赖项了，可以继续学习另一种绑定方式了。下面就用@Provides注解和Provider<T>接口绑定完全由你自己定制的依赖项吧。

3. @Provides和Provider：提供完全定制的对象

你可以用@Provides注解，或者在configure()方法中绑定，以返回一个完全由你自己定制的对象。比如说，你可能想注入一个非常特别的SpreadsheetAgentFinder（微软的Excel电子表格实现）。

注入器会查看所有标记了@Provides注解方法的返回类型，以决定要注入哪个对象。比如在下面的代码中，HollywoodService会用由provideAgentFinder()方法提供的并带有@Provides注解的AgentFinder。

代码清单3-10 AgentFinderModule——使用@Provides

```
public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure(){ }

  @Provides
  AgentFinder provideAgentFinder()    <- 返回注入器需要的类型
  {
    SpreadsheetAgentFinder finder =
                        new SpreadsheetAgentFinder();    <- 创建SpreadsheetAgentFinder的实例并设定具体值
    finder.setType("Excel 97");
    finder.setPath("C:/temp/agents.xls");
    return finder;
  }

}
```

@Provides方法会变得越来越多，为了不把模块类撑爆，你可能要把它们拆分出去建立自己的类。因此，Guice支持JSR-330的Provider<T>接口；如果你还记得第3.2.6节的内容，应该不会忘记T get()方法。当在AgentFinderModule类中通过toProvider方法绑定到AgentFinder-Provider时，就会调用这个方法。代码如下所示：

代码清单3-11　AgentFinderModule——使用Provider接口

```
public class AgentFinderProvider implements Provider<AgentFinder>
{
  @Override
  public AgentFinder get()                              <-- 使用T get()方法
  {
    SpreadsheetAgentFinder finder = new SpreadsheetAgentFinder();
    finder.setType("Excel 97");
    finder.setPath("C:/temp/agents.xls");
    return finder;
  }
}

public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure()
  {
    bind(AgentFinder.class)
      .toProvider(AgentFinderProvider.class);           <-- 绑定provider
  }
}
```

这是最后一个关于绑定的例子。现在你应该能用Guice绑定你需要的依赖项了。但我们还没讨论依赖项的生命周期范围。了解生命周期非常重要，因为如果对象生命周期设置错误的话，它们可能会存在更长时间并占用更多的内存空间。

3.3.3　在Guice中限定注入对象的生命周期

Guice为注入对象提供了不同级别的生命周期。其中最短的是@RequestScope，然后是@SessionScope，还有就是JSR-330规范中定义的@Singleton，也就是应用级别的生命周期。

在代码中有以下几种方式应用依赖项的生命周期：

- 在要注入的类中；
- 作为绑定声明的一部分（比如bind().to().in()）；
- 和@Provides一起使用注解声明。

上面的列表有点儿抽象，我们还是用限定依赖项生命周期的一小段代码来说明上面这些方式究竟是什么意思吧。

1. 限定注入项的生命周期

假设你希望程序的整个生命周期中只用一个SpreadsheetAgentFinder实例。为此需在类声明中设置@Singleton，如下所示：

```
@Singleton
public class SpreadsheetAgentFinder
{
  ...
}
```

用这个方法还有个好处，可以提醒开发人员注入项对线程安全的要求。因为从理论上来说，`SpreadsheetAgentFinder`类可以多次注入，`@Singleton`范围意味着要保证这个类的线程安全性（第4章会讨论线程安全）。

如果你更喜欢在绑定依赖项时声明其生命周期，你就可以这样做。

2. 用BIND()方法设置生命周期

有些开发人员可能喜欢把所有和注入对象相关的规则都放在一起。还记得代码清单3-9中如何绑定“primary”`AgentFinder`吗？在绑定时设置生命周期与此类似，只要在`bind()`方法串后面再加上`.in(<Scope>.class)`就行了。

下面的代码改进了代码清单3-9，在`bind()`方法串后面加上了`in(Session.class)`，使得在会话（session）范围中能够获取“primary”`AgentFinder`对象。

```
public class AgentFinderModule extends AbstractModule
{
  @Override
  protected void configure()
  {
    bind(AgentFinder.class)
      .annotatedWith(Names.named("primary"))
      .to(WebServiceAgentFinder.class)
      .in(Session.class);
  }
}
```

还有最后一种设置注入对象生命周期的方法：与`@Provides`注解联合设置。

3. 设置@Provides对象的生命周期

你可以在`@Provides`注解旁边加上一个生命周期，以定义由该方法所提供对象的生命周期。比如在代码清单3-9中，可以加上`@Request`注解，将最终提供的`SpreadsheetAgentFinder`实例限定在请求（request）范围中。

```
@Provides @Request
AgentFinder provideAgentFinder()
{
  SpreadsheetAgentFinder finder = new SpreadsheetAgentFinder();
  finder.setType("Excel 97");
  finder.setPath("C:/temp/agents.xls");
  return finder;
}
```

Guice也提供了针对Web应用的注入项生命周期（比如Servlet request范围），当然你也可以根据需要实现自己的注入项生命周期。

现在你已经基本掌握了在Guice中用JSR-330注解满足你的依赖注入需求了。其实Guice还提供许多非JSR-330特性，比如它还支持面向方面的编程（AOP），让你可以实现安全性和日志处理

的横切关注点。要了解这些内容，请参考Guice的在线文档和代码示例。

谨慎选择对象的生命周期！

一个成熟的Java开发人员总是会认真考虑对象的生命周期。创建无状态对象相对来说成本低廉，无需考虑其生命周期。JVM会在需要时创建和销毁它们，毫无障碍。（第6章详细讨论了JVM和性能。）

另一方面，状态对象总是需要设定生命周期！你应该认真考虑是否要让一个对象的生命周期贯穿整个应用程序的运行期，或者仅存在于当前会话，还是只在当前请求中。接下来就要考虑对象的线程安全性了（第4章会进一步讨论这个问题）。

3.4　小结

IoC是个复杂的概念。但通过对工厂和服务定位器模式的探讨，你能了解基本IoC实现是如何工作的。工厂模式有助于你理解DI以及DI给代码带来的好处。即便DI范式接受起来非常困难，它也值得你继续坚持，因为它能让你编写松散耦合的代码，让代码更易测试和更易读。

JSR-330不仅仅是统一DI通用功能的重要标准，它还提供了你需要了解的幕后规则及限制。通过研究标准DI注解集，你会更加欣赏不同DI框架对规范的实现，因而可以更有效地使用它们。

Guice是JSR-330的参考实现，同时它也是一个流行的、轻量级的DI框架。实际上，对于很多应用程序来说，使用Guice和与JSR-330兼容的注解集可能就足以满足你对DI的需求了。

如果你是从头开始看这本书的，我们认为你应该稍事休息了！放下手中的书，去做些别的事情，然后再精神饱满地回来继续阅读下一主题——所有优秀的Java开发人员都应该掌握的并发。

第 4 章

现代并发

4

本章内容

- 并发理论
- 块结构并发
- `java.util.concurrent`包
- 用分支/合并框架实现轻量并发
- Java内存模型（JMM）

本章会从基本概念开始，并在块结构并发上做短暂停留。在Java 5之前，这是唯一值得把玩的技术，就是搁在现在，也很值得玩味。之后，我们会介绍每个开发人员都应该了解的`java.util.concurrent`包以及如何使用其提供的基本并发构建块。

我们会以新的分支/合并框架为结尾。看完本章后，你就有能力使用这些新的并发技术了。你还能掌握充分的理论知识，并以此为基础，能够完全理解本书后面讨论到的非Java语言的不同并发视图。

我们不打算在本章把所有的并发知识都讲完，先告诉你一些起步的知识就足够了。另外我们还会告诉你在编写并发代码时需要深入了解些什么，并阻止你在编写代码时涉足险境。但如果你想成为一名真正一流的多线程编程高手，只知道这里讲的知识还不够。这里向你推荐两本专门讨论Java并发编程的书，其中一本是Doug Lea写的《Java并发编程》（第二版，Prentice Hall，1999），另一本是Brian Goetz跟人合著的《Java并发编程实战》（Addison-Wesley Professional，2006）。

但我已经了解线程了！

这是开发人员最常犯的（也许是致命的）错误之一。他们自认为熟悉`Thread`，`Runnable`和语言层面那些原始的基础Java并发机制就是合格的并发编程人员了。实际上，并发是个非常广泛的主题，即便是最好的开发人员，从事多线程开发工作多年，有丰富的经验，要想写出优质的多线程代码也很困难，而且很难保证不出问题。

还有一点你应该注意，目前并发领域正如火如荼地开展着研究工作，这些研究肯定会对你所用到的Java及其他语言产生影响。如果非要我们挑一个在未来五年中很可能会改变行业惯例的计算机基础领域，那就是并发。

本章的目的是让你了解决定Java并发工作方式的底层平台机制。我们还会充分讨论通行的并发理论和词汇，让你理解其中涉及的问题，并教你认识到让并发正确工作的必要性和在这个过程中所遇到的困难。实际上，这里是我们的起点。

4.1　并发理论简介

为了理解在Java中编写并发程序的方法，我们来聊聊相关理论。先讨论一下Java线程模型的基础知识。

之后，我们会讨论系统设计和实现中“设计原则”的影响以及其中最主要的两个原则：安全性和活跃度。我们还会提到其他一些原则，然后讨论这些原则经常相互冲突的原因，以及并发系统中为什么会有开销。

本节的最后我们会看一个多线程系统的例子，并向你证明`java.util.concurrent`是多么自然的编码方法。

4.1.1　解释Java线程模型

Java线程模型建立在两个基本概念之上：

- 共享的、默认可见的可变状态
- 抢占式线程调度

我们从几个侧面思考一下这两个概念。

- 所有线程可以很容易地共享同一进程中的对象。
- 能够引用这些对象的任何线程都可以修改这些对象。
- 线程调度程序差不多任何时候都能在核心上调入或调出线程。
- 必须能调出运行时的方法，否则无限循环的方法会一直占用CPU。

  然而这种不可预料的线程调度可能会导致方法“半途而废”，并出现状态不一致的对象。某一线程对数据做出修改时，会让其他线程无法见到本应可见的修改。为了缓解这些风险，Java提出了最后一点要求。
- 为了保护脆弱的数据，对象可以被锁住。

Java基于线程和锁的并发非常底层，并且一般都比较难用。为了解决这个问题，Java 5引入了一组并发类库`java.util.concurrent`。这个包中提供了一套编写并发代码的工具，很多程序员都觉得它要比传统的块结构并发原语易用。

经验教训

Java是第一个内置多线程编码支持的主流编程语言。这在当时可以说是一个巨大的进步，但15年之后的今天，我们对于如何编写并发代码已经是了若指掌了。

> 事实证明，Java最初的一些设计决策给大多数程序员编写多线程代码带来了很多困难。这的确很糟糕，因为硬件一直朝着多核处理器方向发展，而唯一能利用好这些核心的就是并发代码。本章会讨论一些在编写并发代码时所遇到的困难。现代处理器对并发编程有着合理的需求，我们在第6章讨论性能时还会涉及其中的一些细节。

随着开发人员编写并发代码的经验越来越丰富，他们发现自己所关注的一些重要系统问题一再出现。我们把这些关注点称为“设计原则”——存在于并发OO系统实际设计中的指导性原则（并经常相互冲突）。

在后面几节，我们会花点时间了解一下其中最重要的几个原则。

4.1.2　设计理念

Doug Lea在创造他那里程碑式的作品`java.util.concurrent`时列出了下面这些最重要的设计原则：

- 安全性（也叫做*并发类型安全性*）
- 活跃度
- 性能
- 重用性

下面我们来逐一解读。

1. 安全性与并发类型安全性

安全性是指不管同时发生多少操作都能确保对象保持自相一致。如果一个对象系统具备这一特性，那它就是*并发类型安全的*。

可能你从它的名字就猜出来了，并发可以看做是常规对象建模和类型安全概念的一种延伸。在非并发代码中，要确保不管调用了对象中的什么公开方法，对象最后总是处于一个定义良好并且一致的状态下。通常用来达成这一点的做法是保证对象所有状态都私有，并且开放出来的公开API方法只能以自相一致的方式修改对象状态。

并发类型安全的概念跟对象类型安全一样，但它用在更复杂的环境下。在这样的环境中，其他线程在不同CPU内核上同时操作同一对象。

> **保证安全**
>
> 保证安全的策略之一是在处于非一致状态时绝不能从非私有方法中返回，也绝不能调用任何非私有方法，而且也绝不能调用其他任何对象中的方法。如果把这个策略跟某种对非一致对象的保护办法（比如同步锁或临界区）结合起来，就可以保证系统是安全的。

2. 活跃度

在一个活跃的系统中，所有做出尝试的活动最终或者取得进展，或者失败。

这个定义中的关键词是“最终”——运行中的瞬时故障（尽管不理想，但单独来看这不是问题）和永久故障是不同的。下面这几种底层问题可能会导致系统出现瞬时故障：

- 处于锁定状态或者在等待得到线程锁
- 等待输入（比如网络I/O）
- 资源的暂时故障
- CPU没有足够的空闲时间运行该线程

导致系统出现永久故障的原因较多，其中最常见的是：

- 死锁
- 不可恢复的资源问题（比如NFS不可访问）
- 信号丢失

尽管你对它们可能都已经很熟悉了，但本章后续还是会讨论一下锁定和其他几个问题。

3. 性能

系统性能可以通过几种不同的方式量化。我们会在第6章讨论性能分析和优化技术，并且会介绍一些你应该了解的指标。现在，你可以把性能看成是测量系统用给定资源能做多少工作的办法。

4. 可重用性

可重用性是第四个设计原则，其他它原则中并没涉及这一点。尽管有时不容易实现，但我们还是非常希望能设计出易于重用的并发系统。用可重用工具集（比如`java.util.concurrent`），并把不可重用的应用代码构建在工具集之上是一种可行的办法。

4.1.3　这些原则如何以及为何会相互冲突

设计原则经常相互对立，这种紧张关系使得并发系统的设计很难达到优秀的水准。

- 安全性与活跃度相互对立——安全性是为了确保坏事不会发生，而活跃度要求见到进展。
- 可重用的系统倾向于对外开放其内核，可这会引发安全问题。
- 一个安全但编写方式幼稚的系统性能通常都不会太好，因为里面一般会用大量的锁来保证安全性。

最终应该尽量让代码达到一种平衡的状态，使其能够灵活地适用于各种问题，却又能保证安全性，同时活跃度和性能也可以达到一定水平。这种境界相当高，但你很幸运，我们马上教你一些实战技巧。下面是几个最常见的粗浅办法。

- 尽可能限制子系统之间的通信。隐藏数据对安全性非常有帮助。
- 尽可能保证子系统内部结构的确定性。比如说，即便子系统会以并发的、非确定性的方式进行交互，子系统内部的设计也应该参照线程和对象的静态知识。
- 采用客户端应用必须遵守的策略方针。这个技巧虽然强大，却依赖于用户应用程序的合作程度，并且如果某个糟糕的应用不遵守规则，便很难发现问题所在。
- 在文档中记录所要求的行为。这是最逊的办法，但如果代码要部署在非常通用的环境中，就必须采用这个办法。

开发人员应该了解所有可能的安全机制，而且尽可能采用最强的技术，但同时你也应该知道，在某些情况下只能采用那些比较逊的办法。

4.1.4　系统开销之源

并发系统中的系统开销是与生俱来的，这些开销来自：

- 锁与监测
- 环境切换的次数
- 线程的个数
- 调度
- 内存的局部性[①]
- 算法设计

你应该以此为基础在大脑中列一个检查列表。在编写并发代码时，应该确保自己对列表中的每一项都认真考虑过了，然后再来“搞定”代码。

算法设计

这是一个能让开发人员脱颖而出的领域。无论用什么语言，学习算法设计都能让你成为更好的程序员。在这里我们向你推荐由Thomas H. Corman等人编著的《算法导论》（MIT，2009）和Steven Skiena写的《算法设计手册》（Springer-Verlag，2008）。无论你是想了解单线程算法还是想学习并发算法，它们都是值得阅读的好书。

本章会提到许多系统开销的源头（还有第6章讨论性能的部分）。

4.1.5　一个事务处理的例子

本节前面的内容都太理论化了，所以我们补充一个并发程序设计的例子来实证一下。在这个例子中你将看到如何用`java.util.concurrent`中的高层类完成这个任务。

假设有一个基本事务处理系统。构建这种程序有个简单的标准办法，就是先将业务流程的不同环节对应到应用程序的不同阶段，然后用不同的线程池表示不同的应用阶段，每个线程池逐一接受工作项，在对每个工作项进行一系列的处理后，交给下一个线程池。通常来说，好的设计会让每个线程池所做的处理集中在一个特定功能区内。如图4-1所示。

如果你设计成这样的程序，就可以提高吞吐量，因为可以设计成同时处理几个工作项。比如在检查一个工作项的信用情况时，可以检查另一个工作项的库存。根据应用程序的处理细节不同，甚至可以同时检查多个订单的库存。

① 局部性指的是程序行为的一种规律：在程序运行中的短时间内，程序访问数据位置的集合限于局部范围。局部性有两种基本形式：时间局部性与空间局部性。时间局部性指的是反复访问同一个位置的数据；空间局部性指的是反复访问相邻的数据。——译者注

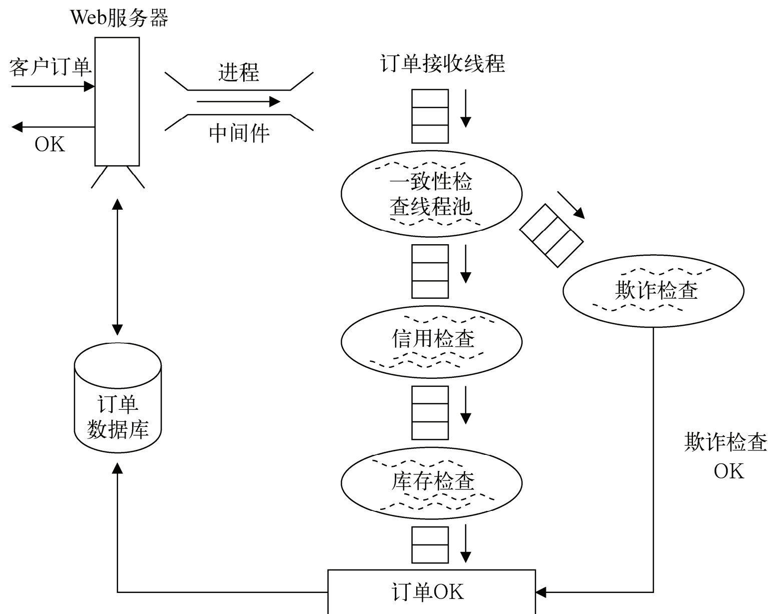


图4-1　多线程应用程序示例

这种设计非常适合用java.util.concurrent包中的类来实现。这个包里有用于执行任务的线程池（Executors类中有一套工厂方法可以创建它们）和在不同线程池之间传递工作的队列，还有并发数据结构（可以用来构建共享缓存，或用于其他用途）和很多其他底层工具。

你可能会问，在Java 5之前，还没有这些类时是怎么办的？一般情况下，开发小组会自己编写并发编程类库，最终会构建出跟java.util.concurrent类似的组件。但这种定制组件大多存在设计缺陷，还会有难以捉摸的并发bug。如果没有java.util.concurrent，开发人员就得重复实现其中的大部分组件（可能会有很多bug，测试也不充分）。

请记住这个例子，我们要转入下一主题——温习一下Java的“传统”并发，并深入了解用它编程困难的原因。

4.2　块结构并发（Java 5 之前）

本章大部分内容都在讨论块同步并发方式的替代方案。如果你想从我们的讨论中获益，就需要深刻理解传统并发的优缺点的重要性。

为此我们要讨论用到synchronized、volatile等并发关键字的那种原始、低级的多线程编程方式。我们把这个讨论放在设计原则的情境中，并且会着眼于下一节将要讨论的内容。

之后我们会简略地解释一下线程的生命周期，然后讨论常见的并发编程技巧和陷阱，比如完全同步的对象，死锁，volatile关键字和不变性。

我们先重温一下同步吧。

4.2.1 同步与锁

你知道的，synchronized既可以用在代码块上也可以用在方法上。它表明在执行整个代码块或方法之前线程必须取得合适的锁。对于方法而言，这意味着要取得对象实例锁（对于静态方法而言则是类锁）。对于代码块，程序员则应该指明要取得哪个对象的锁。

在任何一个对象的同步块或方法中，每次只能有一个线程进入；如果其他线程试图进入，JVM会挂起它们。无论其他线程试图进入的是该对象的同一同步块还是不同的同步块，JVM都会如此处理。这种结构在并发理论中被称为临界区。

注意 你有没有想过Java中用于确立临界区的关键字为什么是synchronized？为什么不是“critical”或“locked”？同步的是什么？我们会在4.2.5节回到这一话题上来，但如果你不知道，或者从来没想过这个问题，你最好花几分钟想一想再继续。

本章要讨论一些比较新的并发技术。但既然说到了同步，我们就顺便看看与Java中的同步和锁相关的一些基本事实吧。希望你对这里的大多数（或全部）知识都已经烂熟于心了。

- 只能锁定对象，不能锁定原始类型。
- 被锁定的对象数组中的单个对象不会被锁定。
- 同步方法可以视同为包含整个方法的同步(this) { ... }代码块（但要注意它们的二进制码表示是不同的）。
- 静态同步方法会锁定它的Class对象，因为没有实例对象可以锁定。
- 如果要锁定一个类对象，请慎重考虑是用显式锁定，还是用getClass()，两种方式对子类的影响不同。
- 内部类的同步是独立于外部类的（要明白为什么会这样，请记住内部类是如何实现的）。
- synchronized并不是方法签名的组成部分，所以不能出现在接口的方法声明中。
- 非同步的方法不查看或关心任何锁的状态，而且在同步方法运行时它们仍能继续运行。
- Java的线程锁是可重入的。也就是说持有锁的线程在遇到同一个锁的同步点（比如一个同步方法调用同一个类内的另一个同步方法）时是可以继续的。

警告 在其他语言中存在不可重入的锁机制（用java也能实现相同的效果，如果你想了解那些让人看了毛骨悚然的详细信息，请参见java.util.concurrent.locks中的ReentrantLock的Javadoc），但和它们打交道太痛苦了，除非你真的知道自己在做什么，否则还是躲之为妙。

对Java同步的温习就到此为止吧。现在我们来看一下线程在其生命周期中的状态变迁。

4.2.2　线程的状态模型

图4-2展示了线程生命周期的发展过程——从创建到运行，到再次运行之前（或被资源阻塞）可能被挂起，再到最终完成。

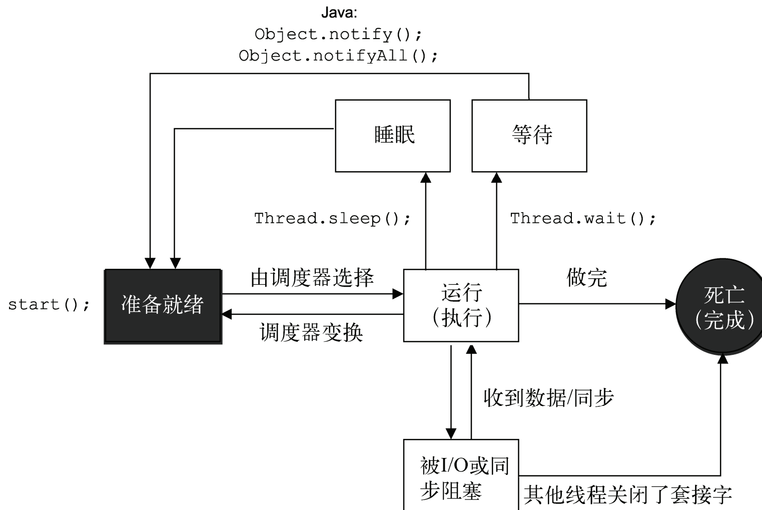


图4-2　Java的线程状态模型

线程最初创建时处于就绪（Ready）状态。然后调度器会找个核心来运行它，如果机器负载过重，那它就可能需要多些时间。开始运行之后，线程通常会消耗掉分配给它的时间，然后回到就绪状态，等到下次再有处理器分配时间片给它。这是我们在4.1.1节提过的抢占式线程调度的标准动作。

除了由调度器发起的标准动作，线程本身也能表明它此时无法使用核心工作。这可能是因为程序代码通过`Thread.sleep()`告诉线程在继续之前应该暂停，或者因为线程必须等待通知（通常需要满足某些外部条件）。这时线程会从核心中移走，并释放它持有的锁。只有通过唤醒才能再次运行线程（在达到睡眠时长之后，或收到了恰当的信号），进入就绪状态。

线程可能会因为等待I/O或等待获取其他线程持有的锁而被阻塞。这时线程并没有被交换出核心，而是仍然处于繁忙状态，等着获取可用的锁或数据。在得到锁或数据之后，线程会继续执行直到它的时间片结束。

我们接下来讨论一个著名的解决同步问题的办法——完全同步对象。

4.2.3　完全同步对象

前面介绍了并发类型安全的概念，还提到了一种用来达成这种安全性的策略（在“保证安全”的边栏中）。现在我们来看一下这个策略更完整的描述，它通常被称为完全同步对象。如果一个

类遵从下面所有规则，就可以认为它是线程安全并且活跃的。

一个满足下面所有条件的类就是完全同步类。

- 所有域在任何构造方法中的初始化都能达到一致的状态。
- 没有公共域。
- 从任何非私有方法返回后，都可以保证对象实例处于一致的状态（假定调用方法时状态是一致的）。
- 所有方法经证明都可在有限时间内终止。
- 所有方法都是同步的。
- 当处于非一致状态时，不会调用其他实例的方法。
- 当处于非一致状态时，不会调用非私有方法。

假定有一个分布式微博工具，代码清单4-1是其后台中的类。在它的propagateUpdate()方法被调用时，ExampleTimingNode类会收到更新，也可以通过查询看它是否收到了特定更新。这是经典的读写操作相互冲突的情景，需要通过同步防止出现不一致状态。

代码清单4-1 完全同步类

```
public class ExampleTimingNode implements SimpleMicroBlogNode {

  private final String identifier;

  private final Map<Update, Long> arrivalTime
  ➥ = new HashMap<>();

  public ExampleTimingNode(String identifier_) {
    identifier = identifier_;
  }

  public synchronized String getIdentifier() {
    return identifier;
  }

  public synchronized void propagateUpdate(
  ➥ Update update_) {
    long currentTime = System.currentTimeMillis();
    arrivalTime.put(update_, currentTime);
  }

  public synchronized boolean confirmUpdateReceived(
  ➥ Update update_) {
    Long timeRecvd = arrivalTime.get(update_);
    return timeRecvd != null;
  }
}
```


这是一个既安全又活跃的类，第一眼看上去让人感觉很了不起。但随之而来的是性能问题，既安全又活跃的东西速度不一定也能很快。必须用synchronized去协调对Map arrivalTime的所有访问（get和put），而这个锁最终会把你的速度拖慢。这是并发处理方式的主要问题。

代码的脆弱性

除了性能问题，代码清单4-1中的代码还很脆弱。你看，它从来不会在同步方法之外去碰arrivalTime，实际上只是调用get和put方法，但这只有在代码量很小的情况下才有可能。在真实的大型系统中，代码太多而无法实现这种方法。同时，bug也很容易潜伏在庞大的代码库中，这也是Java社区开始寻求更完善的解决方法的另一个原因。

4.2.4　死锁

并发的另一个经典问题是死锁。代码清单4-2稍微扩展了一下上个例子。在这一版中，除了记录最近一次更新的时间，每个节点收到更新时还会通知另外一个节点。

这段代码试图构建一个多线程的更新处理系统。注意，这段代码是为了解释死锁，不要把它用到你的工作中。

代码清单4-2　死锁的例子

```
public class MicroBlogNode implements SimpleMicroBlogNode {
  private final String ident;

  public MicroBlogNode(String ident_) {
    ident = ident_;
  }

  public String getIdent() {
    return ident;
  }

  public synchronized void propagateUpdate(Update upd_, MicroBlogNode
     backup_) {
    System.out.println(ident +": recvd: "+ upd_.getUpdateText()
    ➥ +" ; backup: "+backup_.getIdent());
    backup_.confirmUpdate(this, upd_);
  }

  public synchronized void confirmUpdate(MicroBlogNode other_, Update
     update_) {
    System.out.println(ident +": recvd confirm: "+
    ➥ update_.getUpdateText() +" from "+other_.getIdent()k);
  }
}

final MicroBlogNode local =                                  // 关键字final是必需的
➥ new MicroBlogNode("localhost:8888");
final MicroBlogNode other = new MicroBlogNode("localhost:8988");
final Update first = getUpdate("1");
final Update second = getUpdate("2");

new Thread(new Runnable() {
  public void run() {
    local.propagateUpdate(first, other);                     // 第一个更新发送给第一个线程
  }
```

```
}).start();

new Thread(new Runnable() {
  public void run() {
    other.propagateUpdate(second, local);
  }
}).start();
```

第二个更新发送给第二个线程

乍一看，这段代码没什么毛病。有两个更新分别发送给不同的线程，每个都必须由后备线程进行确认。这看起来不是什么离奇古怪的设计——如果一个线程失效，另外一个线程还可以挑起重担。

如果你运行这段代码，一般都会碰到死锁——两个线程都说自己收到了更新，但它俩谁都不会以备份线程的身份确认收到了更新。因为每个线程在确认方法能够确认之前都要求另外一个线程释放线程锁，如图4-3所示。

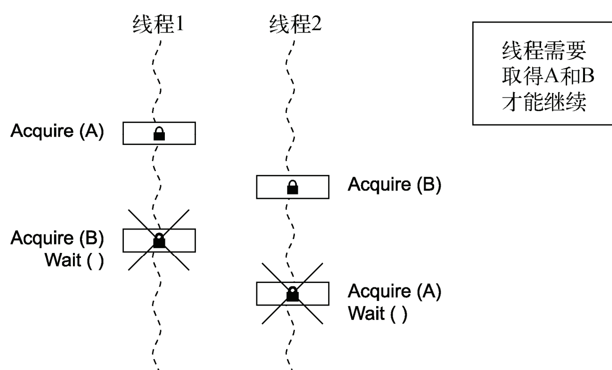


图4-3　死锁线程

有一个处理死锁的技巧，就是在所有线程中都以相同的顺序获取线程锁。在前例中，第一个线程以A、B的顺序获取锁，而第二个线程获取锁的顺序是B、A。如果两个线程都用A、B的顺序，死锁的情况就可以避免，因为第二个线程在第一个线程完成并释放锁之前会一直被阻塞住。

就完全同步对象方式而言，要防止这种死锁出现是因为代码破坏了状态一致性规则。当有消息到达时，接受节点会在消息处理过程中调用另外一个对象——它发起这个调用时状态是不一致的。

接下来，我们会返回来解释前面抛出的那个问题：为什么Java中用来标识临界区的关键字是`synchronized`？这会引导我们转而讨论不可变性和关键字`volatile`。

4.2.5　为什么是synchronized

最近几年并发编程变化最大的是硬件领域。在以前，程序员可能常年累月都碰不到需要支持多处理器核心（两个或最多三个）的系统。因此并发编程过去主要考虑如何分享CPU时间——线程们在单核上轮流上位，相互调换。

现如今，任何比手机大点儿的东西都是多核的，所以我们的认知模型也该换换了，应该把多个线程在同一物理时刻运行在不同核心（并且很可能会操作共享的数据）的情况也考虑在内。如图4-4所示。为了提高效率，同时运行的每个线程可能都会有它正在处理的数据的缓存复本。记住这幅图，让我们回到选择用什么关键字来表示被锁定的代码块或方法这个问题上。

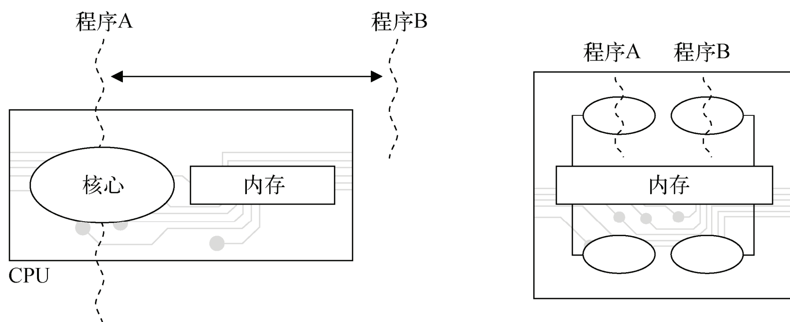


图4-4　考虑并发和线程的新、老方式

我们在前面问过，代码清单4-1中被同步的是什么？答案是：*被同步的是在不同线程中表示被锁定对象的内存块*。也就是说，在`synchronized`代码块（或方法）执行完之后，对被锁定对象所做的任何修改全部都会在线程锁释放之前刷回到主内存中，如图4-5所示：

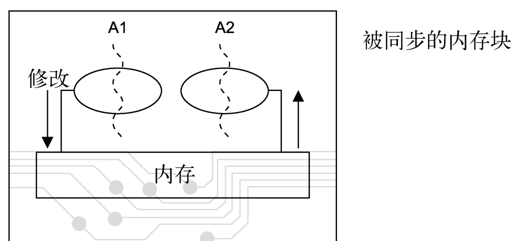


图4-5　不同线程对一个对象的修改通过主内存传播

另外，当进入一个同步的代码块，得到线程锁之后，对被锁定对象的任何修改都是从主内存中读出来的，所以在锁定区域代码开始执行之前，持有锁的线程就和锁定对象主内存中的视图同步了。

4.2.6　关键字`volatile`

Java在其混沌初开的时期（Java 1.0）就已经把`volatile`作为关键字了，它是一种简单的对象域同步处理办法，包括原始类型。一个`volatile`域需遵循如下规则：

- 线程所见的值在使用之前总会从主内存中再读出来。
- 线程所写的值总会在指令完成之前被刷回到主内存中。

可以把围绕该域的操作看成是一个小小的同步块。程序员可以借此编写简化的代码，但付出的代价是每次访问都要额外刷一次内存。还有一点要注意，volatile变量不会引入线程锁，所以使用volatile变量不可能发生死锁。

更加微妙的是，volatile变量是真正线程安全的，但只有写入时不依赖当前状态（读取的状态）的变量才应该声明为volatile变量。对于要关注当前状态的变量，只能借助线程锁保证其绝对安全性。

4.2.7　不可变性

不可变对象的应用是十分有价值的技术。这些对象或没有状态，或只有final域（因此只能在构造方法中赋值）。它们总是安全而又活跃的。它们的状态不能修改，所以不可能出现不一致的情况。

可这样对象初始化的所有值都必须传入构造方法。这会导致构造方法的参数很多，看起来又蠢又笨。因此很多程序员选择工厂方法FactoryMethod代替构造方法。工厂方法很简单，就是类中的一个静态方法，用来代替构造方法创建新对象。此时构造方法通常被声明为protected或private的，从而使工厂方法成为实例化对象的唯一办法。

但是还存在要将众多参数传入FactoryMethod的问题。有时候这不太方便，尤其是初始化对象所需的状态参数有多个不同来源时。

构建器模式可以解决这个问题。它由两部分组成：一个是实现了构建器泛型接口的内部静态类，另一个是构建不可变类实例的私有构造方法。

内部静态类是不可变类的构建器，开发人员只能通过它获取不可变类的新实例。比较常见的实现方式是让构建器类拥有与不可变类一模一样的域，但构建器的域是可修改的。

下面这段代码展示了如何建立不可变的微博更新模型（根据本章前面的例子所构建）。

代码清单4-3　不可变对象及构建器

```
public interface ObjBuilder<T> {
  T build();                                    // 构建器接口
}

public class Update {
  private final Author author;                  // 必须在构造方法中初始化final域
  private final String updateText;

  private Update(Builder b_) {
    author = b_.author;
    updateText = b_.updateText;
  }

  public static class Builder
  ➥ implements ObjBuilder<Update> {            // 构造器类必须是静态内部类
    private Author author;
    private String updateText;
```

```
      public Builder author(Author author_) {        可用在调用链中返回
        author = author_;                            Builder的方法
        return this;
      }

      public Builder updateText(String updateText_) {
        updateText = updateText_;
        return this;
      }
      public Update build() {
        return new Update(this);                     略去hashCode()和
      }                                              equals()方法
    }

}
```

有了这段代码，你就可以创建新的Update对象：

```
Update.Builder ub = new Update.Builder();
Update u = ub.author(myAuthor).updateText("Hello").build();
```

这是一个得到广泛应用的通用模式。实际上，在代码清单4-1和4-2中我们已经使用了不可变对象的特性。

关于不可变对象的最后一点——关键字final仅对其直接指向的对象有用。如图4-6所示，对主对象的引用不能赋值为对象3，但在主对象内部，对1的引用可以改为指向对象2。也就是说final引用可以指向带有非final域的对象。

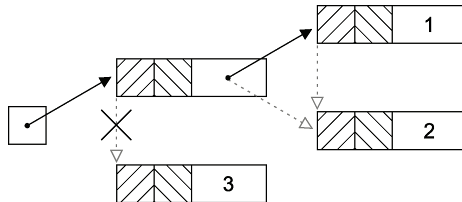


图4-6　值的不可变性与引用

不可变是非常强的技术，用处十分广泛。但有时候只用不可变对象开发效率不行，因为每次修改对象状态就需要构建一个新对象。所以可变对象很有必要保留。

我们马上就要开始讨论本章最重要的主题——java.util.concurrent中更加现代化、概念更简单的并发API。看看怎么用它们写代码。

4.3　现代并发应用程序的构件

随着Java 5的到来，Java对并发的重新思考也浮出了水面。这些新思想主要体现在java.util.concurrent包上，其中包含了大量用来编写多线程代码的新工具。在后续版本中，这些工具不断得到改进，但其工作方式却依然保持不变，并且直到今天还是对开发人员很有帮助。

我们马上快速过一下java.util.concurrent中主要的类及相关包，比如atomic和locks包。我们会向你介绍这些类及其适用的情景。你也应该读一下它们的Javadoc，并尝试熟悉整个包——它们使编写并发类容易多了。

代码迁移

如果你还有基于（Java 5之前的）老办法编写的多线程代码，建议你用java.util.concurrent重构。按我们的经验，几乎在所有案例中，如果你特意把代码迁移到新的API中，代码就会得以改进。你的努力付出将使代码在清晰性和可靠性上得到极大提升。

请把这次讨论当做并发编程的启动工具，而不是一次研讨会。想要充分利用好java.util.concurrent，你还需要知道更多的知识。

4.3.1 原子类：**java.util.concurrent.atomic**

java.util.concurrent.atomic中有几个名字以Atomic打头的类。它们的语义基本上和volatile一样，只是封装在一个API里了，这个API包含为操作提供的适当的原子（要么不做，要做就全做）方法。对于开发人员来说，这是非常简单的避免在共享数据上出现竞争危害[①]的办法。

在编写这些实现时利用了现代处理器的特性，所以如果能从硬件和操作系统上得到适当的支持，它们可以是非阻塞（无需线程锁）的，而大多数现代系统都能提供这种支持。常见的用法是实现序列号机制，在AtomicInteger或AtomicLong上用原子操作getAndIncrement()方法。

要做序列号，该类应该有个nextId()方法，每次调用时肯定能返回一个唯一并且完全增长的数值。这和数据库里序列号的概念很像（所以这个变量叫这个名字）。

来看一段产生序列号的代码：

```
private final AtomicLong sequenceNumber = new AtomicLong(0);

public long nextId() {
  return sequenceNumber.getAndIncrement();
}
```

注意 原子类不是从有相似名称的类继承而来的，所以AtomicBoolean不能当Boolean用，AtomicInteger也不是Integer，虽然它确实扩展了Number。

接下来，我们会检查一下java.util.concurrent如何对同步模型的核心建模——Lock接口。

4.3.2 线程锁：**java.util.concurrent.locks**

块结构同步方式基于锁这样一个简单的概念。这种方式有几个缺点。

- 锁只有一种类型。
- 对被锁住对象的所有同步操作都是一样的作用。

① 竞争危害（race hazard）又名竞态条件（race condition）。一个系统或进程的输出，依赖于不受控制事件的出现顺序或时机。例如两个进程都试图修改一个共享内存的内容。在没有并发控制的情况下，最后的结果取决于两个进程的执行顺序与时机，如果发生了并发访问冲突，最后的结果是不正确的。——译者注

- 在同步代码块或方法开始时取得线程锁。
- 在同步代码块或方法结束时释放线程锁。
- 线程或者得到锁，或者阻塞——没有其他可能。

如果我们要重构对线程锁的支持，有几处可以得到提升。

- 添加不同类型的锁，比如读取锁和写入锁。
- 对锁的阻塞没有限制，即允许在一个方法中上锁，在另一个方法中解锁。
- 如果线程得不到锁，比如锁由另外一个线程持有，就允许该线程后退或继续执行，或者做点别的事情——运用tryLock()方法。
- 允许线程尝试取锁，并可以在超过等待时间后放弃。

能实现以上这些的关键就是java.util.concurrent.locks中的Lock接口。还有它的两个实现类。

- ReentrantLock——本质上跟用在同步块上那种锁是一样的，但它要稍微灵活点儿。
- ReentrantReadWriteLock——在需要读取很多线程而写入很少线程时，用它性能会更好。

块结构并发能实现的所有功能都可以用Lock接口实现。下面是用ReentrantLock重写的那个死锁的例子。

代码清单4-4　用ReentrantLock重写死锁

```
private final Lock lock = new ReentrantLock();

public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  lock.lock();                                         // 每个线程都先锁住自己的锁
  try {
    System.out.println(ident +": recvd: "+
    ➥ upd_.getUpdateText() +" ; backup: "+
    ➥ backup_.getIdent());
      backup_.confirmUpdate(this, upd_);               // 调用confirmUpdate()知悉其他线程
  } finally {
    lock.unlock();
  }
}
public void confirmUpdate(MicroBlogNode other_, Update upd_) {
  lock.lock();                                         // ❶ 尝试锁住其他线程
  try{
    System.out.println(iden +": recvd confirm: "+
    ➥ upd_.getUpdateText() +" from "+ other_.getIdentifier());
  } finally {
    lock.unlock();
  }
}
```

锁住其他线程的尝试❶通常都会失败，因为它已经被锁住了（如图4-3所示）。这就是导致死锁出现的原因。

用锁时带上try...finally

把lock()放在try...finally块中（释放也在这里）的模式是另外一个好用的小工具。在跟块结构并发相似的情景中它同样很好用。而另一方面，如果需要传递Lock对象，比如从一个方法中返回，则不能用这个模式。

使用Lock对象可能要比块结构方式强大得多，但有时用它们很难设计出完善的锁定策略。

对付死锁的策略有很多，但你应该特别注意一个不起任何作用的策略。请看下面这段代码中新版的propagateUpdate()方法（假定confirmUpdate()也做出了同样的修改）。在这个例子中，我们用带有超时机制的tryLock()替换了无条件的锁。通过这种办法可以为其他线程提供得到线程锁的机会，从而去除死锁。

代码清单4-5　一次有缺陷的解决死锁问题的尝试

```
public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  boolean acquired = false;

  while (!acquired) {
    try {
      int wait = (int)(Math.random() * 10);
      acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);    ◁ 尝试与锁定，超时时长随机
      if (acquired) {
        System.out.println(ident +": recvd: "+
        ➥ upd_.getUpdateText() +" ; backup: "+backup_.getIdent());
        backup_.confirmUpdate(this, update_);    ◁ 在其他线程上确认
      } else {
        Thread.sleep(wait);
      }
    } catch (InterruptedException e) {
    } finally {
      if (acquired) lock.unlock();    ◁ 仅在锁定时解锁
    }
  }
}
```

如果运行代码清单4-5中的代码，你会发现它有时候还是不能解决死锁问题。你能看到“received confirm of update”，但它并不会一直出现，时有时无。

实际上，死锁问题并没有真正解决，因为如果线程取得了第一个锁（在propagateUpdate()中），它才会调用confirmUpdate()，并且在完成之前绝不会释放第一个锁。即使两个线程都能在彼此调用confirmUpdate()之前取得第一个线程锁，它们还是会产生死锁。

如果取得第二个锁的尝试失败，能真正解决问题的办法是让线程释放其持有的第一个锁，再次从头开始等待，从而使其他线程有机会得到完整的锁集合，能走完全程。代码如下所示。

代码清单4-6　修正死锁

```
public void propagateUpdate(Update upd_, MicroBlogNode backup_) {
  boolean acquired = false;
  boolean done = false;
```

```
    while (!done) {
      int wait = (int)(Math.random() * 10);
      try {
        acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);
        if (acquired) {
          System.out.println(ident +": recvd: "+
          ➥ upd_.getUpdateText() +" ; backup: "+backup_.getIdent());
          done = backupNode_.tryConfirmUpdate(this, update_);   <-- 检查tryConfirmUpdate()的返回值
        }
      } catch (InterruptedException e) {
      } finally {
        if (acquired) lock.unlock();
      }
      if (!done) try {
        Thread.sleep(wait);
      } catch (InterruptedException e) { }   <-- 如果done为false，释放锁并等待
    }
  }

  public boolean tryConfirmUpdate(MicroBlogNode other_, Update upd_) {
    boolean acquired = false;
    try {
      int wait = (int)(Math.random() * 10);
      acquired = lock.tryLock(wait, TimeUnit.MILLISECONDS);

      if (acquired) {
        long elapsed = System.currentTimeMillis() - startTime;
        System.out.println(ident +": recvd confirm: "+
        ➥ upd_.getUpdateText() +" from "+other_.getIdent()
        ➥ +" - took "+ elapsed +" millis");
        return true;
      }
    } catch (InterruptedException e) {
    } finally {
      if (acquired) lock.unlock();
    }

    return false;
  }
```

这一版会检查`tryConfirmUpdate()`的返回码。如果为`false`，最初的锁被释放。该线程会暂停一段时间，让其他线程有机会获取锁。

把这段代码运行几次，你会发现这两个线程基本上总能走完全程——死锁问题已经被你解决了。你也许想试验试验之前版本中那段代码的不同形式，诸如最原始的、有缺陷的或被改正的。通过对这些代码的演练，你能对锁机制有更深刻的理解，并且开始渐渐地凭直觉避免死锁问题的出现。

为什么那个有缺陷的版本有时候能奏效？

你已经看到了，死锁仍然存在，那是什么原因导致这个版本中的代码有时可以成功呢？代码中附加的复杂性是罪魁祸首。它影响JVM的线程调度器，让它变得更加难以预测。这意味着它有时候能让某个线程（通常是第一个）在其他线程运行之前进入`confirmUpdate()`方法并取得第二个锁。这种情况也会发生在原始代码中，只是可能性更低罢了。

我们只是揭开了Lock各种可能性的面纱——有很多种方法可以产生更加复杂的锁定结构。接下来我们就来讨论其中一个概念——锁存器。

4.3.3 CountDownLatch

CountDownLatch是一种简单的同步模式，这种模式允许线程在通过同步屏障之前做些少量的准备工作。

为了达到这种效果，在构建新的CountDownLatch实例时要给它提供一个int值（计数器）。此外，还有两个用来控制锁存器的方法：countDown()和await()。前者对计数器减1，而后者让调用线程在计数器到0之前一直等待。如果计数器已经为0或更小，则它什么也不做。这个简单的机制使得这种所需准备最少的模式非常容易部署。

在下面的代码中，同一进程内的一组更新处理线程至少必须有一半线程正确初始化（假定更新处理线程的初始化要占用一定时间）之后，才能开始接受系统发送给它们中的任何一个线程的更新。

代码清单4-7 用锁存器辅助初始化

```
public static class ProcessingThread extends Thread {
  private final String ident;
  private final CountDownLatch latch;

  public ProcessingThread(String ident_, CountDownLatch cdl_) {
    ident = ident_;
    latch = cdl_;
  }
  public String getIdentifier() {
    return identifier;
  }
  public void initialize() {                    <-- 节点初始化
    latch.countDown();
  }
  public void run() {
    initialize();
  }
}

final int quorum = 1 + (int)(MAX_THREADS / 2);
final CountDownLatch cdl = new CountDownLatch(quorum);

final Set<ProcessingThread> nodes = new HashSet<>();
try {
  for (int i=0; i<MAX_THREADS; i++) {
    ProcessingThread local = new ProcessingThread("localhost:"+
    ➥ (9000 + i), cdl);
    nodes.add(local);
    local.start();
  }
  cdl.await();                                   <-- 达到quorum，开始发送更新
} catch (InterruptedException e) {
} finally {
}
```

这段代码把锁存器的值设置为quorum。一旦被初始化的线程达到这个数量，就可以开始处理更新了。每个线程完成初始化后都会马上调用countDown()，所以主线程只需等待quorum的到来，然后启动（并派发更新，尽管我们没给出那部分代码）。

我们接下来要讨论的是对多线程开发人员来说最有用的类之一：java.util.concurrent中的ConcurrentHashMap。

4.3.4 ConcurrentHashMap

ConcurrentHashMap类是标准HashMap的并发版本。它改进了Collections类中提供的synchronizedMap()功能，因为那些方法返回的集合中包含的锁要比需要的多。

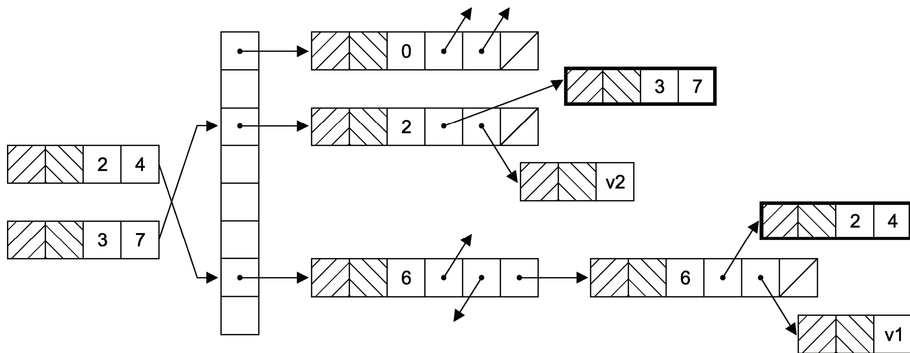


图4-7　HashMap的经典视图

如图4-7所示，传统的HashMap用hash函数来确定存放键/值对的“桶”，这是该类名字中“Hash”的由来。这意味着多线程处理可以更加简单直接——修改HashMap时并不需要把整个结构都锁住，只要锁住即将修改的桶就行了。

提示　好的并发HashMap实现在读取时不用锁，写入时只需锁住要修改的桶。Java基本上能达到这个标准，但这里还有一些大多数开发人员都无需过多关注的底层细节。

ConcurrentHashMap类还实现了ConcurrentMap接口，有些提供了原子操作的新方法。

- putIfAbsent()——如果还没有对应键，则把键/值对添加到HashMap中。
- remove()——如果对应键存在，且值也与当前状态相等（equal），则用原子方式移除键值对。
- replace()——API为HashMap中原子替换的操作方法提供了两种不同的形式。

比如说，如果你把代码清单4-1中的私有final域arrivalTime的类型从HashMap改成ConcurrentHashMap，那就可以把synchronized方法替换成常规的非同步访问。注意代码清单4-8中锁的缺失——根本就没有显式的同步。

代码清单4-8　使用ConcurrentHashMap

```
public class ExampleMicroBlogTimingNode implements SimpleMicroBlogNode {
  ...
private final Map<Update, Long> arrivalTime =
➥ new ConcurrentHashMap <>();
  ...
public void propagateUpdate(Update upd_) {
    arrivalTime.putIfAbsent(upd_, System.currentTimeMillis());
  }
  public boolean confirmUpdateReceived(Update upd_) {
    return arrivalTime.get(upd_) != null;
  }
}
```

4

ConcurrentHashMap是java.util.concurrent包中最有用的类之一。它不仅提供了多线程的安全性，并且性能更优，在日常使用中没有严重的缺陷。接下来我们会讨论它的最佳拍档，用于List的CopyOnWriteArrayList。

4.3.5　CopyOnWriteArrayList

从名字就能看出来，CopyOnWriteArrayList是标准ArrayList的替代品。CopyOnWrite-ArrayList通过增加写时复制（copy-on-write）语义来实现线程安全性，也就是说修改列表的任何操作都会创建一个列表底层数组的新复本（如图4-8所示）。这就意味着所有成形的迭代器[①]都不用担心它们会碰到意料之外的修改。

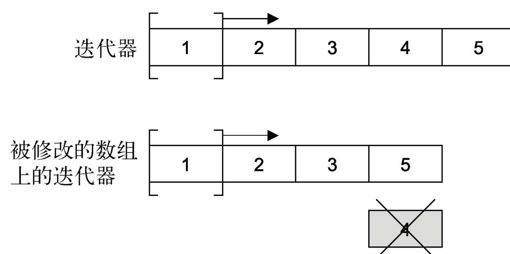


图4-8　写时复制数组

当快速、一致的数据快照（不同的读取器读到的数据偶尔可能会不一样）比完美的同步以及性能上的突破更重要时，这种共享数据的方法非常理想，并经常出现在非关键任务中。

我们来看一个写时复制的案例。假设有个微博的时间线更新，这是一个典型的非关键任务的例子。每个读取器的性能、自身一致性的快照要比全局的一致性更受欢迎。代码清单4-9表示每个用户时间线视图的持有者类。我们将会在代码清单4-10中用它来演示写时复制操作是如何进行的。

① 迭代器（iterator）是一个对象，它的工作是遍历并选择序列中的对象，而客户端程序员不必知道或关心该序列底层的结构（也就是不同容器的类型）。——译者注

代码清单4-9　写时复制案例

```
public class MicroBlogTimeline {
  private final CopyOnWriteArrayList<Update> updates;
  private final ReentrantLock lock;
  private final String name;
  private Iterator<Update> it;                    // 构造方法已省略

  public void addUpdate(Update update_) {
    updates.add(update_);
  }
  public void prep() {                            // 设置迭代器
    it = updates.iterator();
  }
  public void printTimeline() {
    lock.lock();                                  // 需要在这里锁定
    try {
      if (it != null) {
        System.out.print(name+ ": ");
        while (it.hasNext()) {
          Update s = it.next();
          System.out.print(s+ ", ");
        }
        System.out.println();
      }
    } finally {
      lock.unlock();
    }
  }
}
```

我们专门设计了这个类来阐明在写时复制语义下的迭代器行为。你需要在输出方法中锁定，以防止输出在两个线程间乱掉，此外你也能看到两个线程各自的状态。

你可以从下面的代码中调用MicroBlogTimeline类。

代码清单4-10　揭示写时复制行为

```
final CountDownLatch firstLatch = new CountDownLatch(1);
final CountDownLatch secondLatch = new CountDownLatch(1);
final Update.Builder ub = new Update.Builder();

final List<Update> l = new CopyOnWriteArrayList<>();     // ❶ 设置初始状态
l.add(ub.author(new Author("Ben")).updateText("I like pie").build());
l.add(ub.author(new Author("Charles")).updateText(
➥ "I like ham on rye").build());

ReentrantLock lock = new ReentrantLock();
final MicroBlogTimeline tl1 = new MicroBlogTimeline("TL1", l, lock);
final MicroBlogTimeline tl2 = new MicroBlogTimeline("TL2", l, lock);

Thread t1 = new Thread() {
  public void run() {
    l.add(ub.author(new Author("Jeffrey")).updateText(
    ➥ "I like a lot of things").build());
    tl1.prep();
```

```
      firstLatch.countDown();
      try { secondLatch.await(); }
      ➥ catch (InterruptedException e) {  }
      tl1.printTimeline();
    }
  };

  Thread t2 = new Thread(){
    public void run(){
      try {
        firstLatch.await();
        l.add(ub.author(new Author("Gavin")).updateText(
        ➥ "I like otters").build());
        tl2.prep();
        secondLatch.countDown();
      } catch (InterruptedException e) { }
      tl2.printTimeline();
  }
  };
  t1.start();
  t2.start();
```

用锁存器严格限制事件的顺序

用锁存器严格限制事件的顺序

这段代码里有很多辅助的测试代码。但也有很多值得注意的地方：

- CountDownLatch用来严格控制两个线程之间发生的事情。
- 如果用普通的List代替CopyOnWriteArrayList，结果会导致出现ConcurrentModificationException异常。
- 这也是在两个线程之间共享一个Lock对象以控制对共享资源（即STDOUT）访问的例子。如果用块结构方式写这段代码，会显得更加杂乱。

这段代码的输出如下：

```
TL2: Update [author=Author [name=Ben], updateText=I like pie, createTime=0],
     Update [author=Author [name=Charles], updateText=I like ham on rye,
     createTime=0], Update [author=Author [name=Jeffrey], updateText=I like a
     lot of things, createTime=0], Update [author=Author [name=Gavin],
     updateText=I like otters, createTime=0],

TL1: Update [author=Author [name=Ben], updateText=I like pie, createTime=0],
     Update [author=Author [name=Charles], updateText=I like ham on rye,
     createTime=0], Update [author=Author [name=Jeffrey], updateText=I like a
     lot of things, createTime=0],
```

第二行输出（标签为TL1）漏掉了最后一个更新，就是提到了水獭的那个，尽管按锁存器的意思在列表被修改后tl1[①]是可以访问的。这说明了tl1中所包含的迭代器被tl2复制，并且最后一个更新对tl1是不可见的。这就是我们想要展示的写时复制特性。

CopyOnWriteArrayList的性能

使用CopyOnWriteArrayList类要比使用ConcurrentHashMap多花点心思，它是HashMap的即用型并发替代品。这是因为性能问题——写时复制特性意味着如果列表在被读取

① 原文为mbex1，下文同。——译者注

或遍历时做了修改，那就必须复制整个数组。

也就是说如果对列表的修改次数跟读取次数相差不多，这种方式未必能达到较好的性能。但就像我们在第6章一再提到的那样，得到性能优异的代码的唯一可靠的方法就是测试，再测试，并衡量结果。

下一个在并发代码中常用的构件是java.util.concurrent中的Queue。它用于在线程之间切换工作元素，并且还是很多灵活可靠的多线程设计的基础。

4.3.6 Queue

队列是一个非常美妙的抽象概念。不，之所以这么说并不是因为我们生活在伦敦这个世界排队之都。为把处理资源分发给工作单位（或者把工作单元分配给处理资源，这取决于你看待问题的方式），队列提供了一种简单又可靠的方式。

Java中有些多线程编程模式在很大程度上都依赖于Queue实现的线程安全性，所以很有必要充分认识它。Queue接口被放在了java.util包中，因为即便在单线程编程中它也是一个重要的模式，但我们的重点是多线程编程，并且假定你已经熟悉队列的基本用法了。

队列经常用来在线程之间传递工作单元，这个模式通常适合用Queue最简单的并发扩展BlockingQueue来实现。接下来我们就会重点介绍它。

1. BlockingQueue

BlockingQueue还有两个特性。

- 在向队列中put()时，如果队列已满，它会让放入线程等待队列腾出空间。
- 在从队列中take()时，如果队列为空，会导致取出线程阻塞。

这两个特性非常有用，因为如果一个线程（或线程池）的能力超过了其他线程，比较快的线程就会被强制等待，因此可以对整个系统起到调节作用，如图4-9所示。

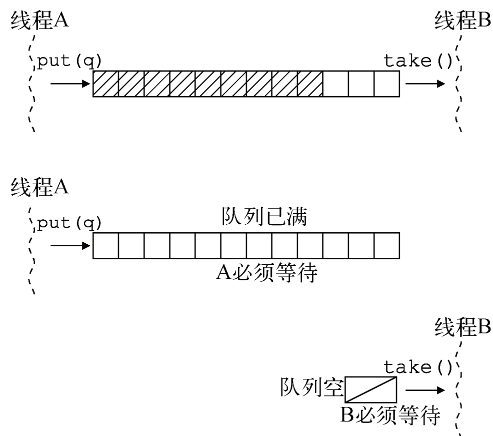


图4-9　BlockingQueue

BlockingQueue的两个实现

Java提供了BlockingQueue接口的两个基本实现：LinkedBlockingQueue和ArrayBlockingQueue。它们的特性稍有不同；比如说，在已知队列的大小而能确定合适的边界时，用ArrayBlockingQueue非常高效，而LinkedBlockingQueue在某些情况下则会快一点儿。

2. 使用工作单元

Queue接口全都是泛型的——它们是Queue<E>，BlockingQueue<E>，等等依此类推。尽管看起来奇怪，但有时候利用这一点把工作项封装在一个人工容器类内却是明智之举。

比如说，你有一个表示工作单元的MyAwesomeClass类，想要用多线程方式处理，与其用BlockingQueue<MyAwesomeClass>不如用BlockingQueue<WorkUnit<MyAwesomeClass>>。其中WorkUnit（或QueueObject，或随你怎么命名这个容器类）是像下面这样的包装接口或类：

```
public class WorkUnit<T> {
  private final T workUnit;

  public T getWork(){ return workUnit; }

  public WorkUnit(T workUnit_) {
    workUnit = workUnit_;
  }
}
```

有了这层间接引用，不用牺牲所包含类型（在此即MyAwesomeClass）在概念上的完整性就可以在这里添加额外的元数据了。

这特别有用。能用上额外元数据的用例很多，下面举几个例子：

- 测试（比如展示一个对象的修改历史）
- 性能指标（比如到达时间或服务质量）
- 运行时系统信息（比如MyAwesomeClass实例是如何被排到队列中的）

以后再在这种间接引用里增加元数据可能会非常困难。如果你发现在某些情况下需要更多的元数据，那么要把它们加入到间接引用中可能需要大量的重构工作，而加在WorkUnit类中就只是个简单的修改。

3. 一个BlockingQueue的例子

我们用一个简单的例子——等着看医生的宠物们——来看看如何使用BlockingQueue。这个例子中有一个等着让医生给做检查的宠物集合。

代码清单4-11 在Java中对宠物建模

```
public abstract class Pet {
  protected final String name;

  public Pet(String name) {
    this.name = name;
  }
```

```
  public abstract void examine();
}

public class Cat extends Pet {
  public Cat(String name) {
    super(name);
  }
  public void examine(){
    System.out.println("Meow!");
  }
}

public class Dog extends Pet
  public Dog(String name) {
    super(name);
  }
  public void examine(){
    System.out.println("Woof!");
  }
}

public class Appointment<T> {
  private final T toBeSeen;

  public T getPatient(){ return toBeSeen; }

  public Appointment(T incoming) {
    toBeSeen = incoming;
  }
}
```

在这个简单的例子中，我们用LinkedBlockingQueue<Appointment<Pet>>表示兽医的候诊队列，Appointment起到了WorkUnit的作用。

兽医对象是由一个队列和一个暂停时间构建的，其中队列是由一个代表接待员的对象提供的预约队列，暂停时间表示兽医在预约之间的停工时间。

我们可以在下面这段代码中建立兽医的模型。在线程运行时，它在一个无限循环中重复调用seePatient()。当然，现实世界中的兽医不可能这样，因为他们晚上和周末要回家，不能一直在办公室等着生病的小动物上门就医。

代码清单4-12 对兽医建模

```
public class Veterinarian extends Thread {
  protected final BlockingQueue<Appointment<Pet>> appts;
  protected String text = "";
  protected final int restTime;
  private boolean shutdown = false;

  public Veterinarian(BlockingQueue<Appointment<Pet>> lbq, int pause) {
    appts = lbq;
    restTime = pause;
  }

  public synchronized void shutdown(){
    shutdown = true;
  }
```

```
    @Override
    public void run(){
      while (!shutdown) {
        seePatient();
        try {
          Thread.sleep(restTime);
        } catch (InterruptedException e) {
          shutdown = true;
        }
      }
    }

    public void seePatient() {
      try {
        Appointment<Pet> ap = appts.take();        <-- 阻塞take
        Pet patient = ap.getPatient();
        patient.examine();
      } catch (InterruptedException e) {
        shutdown = true;
      }
    }
  }
```

在seePatient()方法中，线程会从队列中取出预约，并挨个检查对应的宠物，如果当前队列中没有预约等待，则会阻塞。

4. **BlockingQueue**的细粒度控制

除了简单的take()和offer()API，BlockingQueue还提供了另外一种与队列交互的方式，这种方式对队列的控制力度更大，但稍微有点复杂。这就是带有超时的放入或取出操作，它允许线程在遇到问题时可以从与队列的交互中退出来，转而做点儿其他的事情。

实际上，这个功能并不常用，但它偶尔能派上大用场，所以我们要介绍一下。下面的例子还是来自微博。

代码清单4-13 BlockingQueue行为的例子

```
public abstract class MicroBlogExampleThread extends Thread {
  protected final BlockingQueue<Update> updates;
  protected String text = "";
  protected final int pauseTime;
  private boolean shutdown = false;

  public MicroBlogExampleThread(BlockingQueue<Update> lbq_, int pause_) {
    updates = lbq_;
    pauseTime = pause_;
  }

  public synchronized void shutdown(){
    shutdown = true;                          <-- 使线程可以彻底地结束
  }

  @Override
  public void run(){
    while (!shutdown) {                       <--
      doAction();
```

```
        try {
          Thread.sleep(pauseTime);
        } catch (InterruptedException e) {
          shutdown = true;
        }
      }
    }
    public abstract void doAction();
  }
```

由子类实现具体动作

```
  final Update.Builder ub = new Update.Builder();
  final BlockingQueue<Update> lbq = new LinkedBlockingQueue<>(100);

  MicroBlogExampleThread t1 = new MicroBlogExampleThread(lbq, 10) {
    public void doAction(){
      text = text + "X";
      Update u = ub.author(new Author("Tallulah")).updateText(text).build();
      boolean handed = false;
      try {
        handed = updates.offer(u, 100, TimeUnit.MILLISECONDS);
      } catch (InterruptedException e) {
      }
      if (!handed) System.out.println(
      ➥"Unable to hand off Update to Queue due to timeout");
    }
  };

  MicroBlogExampleThread t2 = new MicroBlogExampleThread(lbq, 1000) {
    public void doAction(){
      Update u = null;
      try {
        u = updates.take();
      } catch (InterruptedException e) {
        return;
      }
    }
  };
  t1.start();
  t2.start();
```

运行这段代码展示了填充队列的速度有多么快，也表明供给线程的速度超过了提取线程的速度。很快，“`Unable to hand off Update to Queue due to timeout`”消息就出现了。

这是“相连线程池”中的一种典型的极端状况，当上游的线程池比下游的快，这种情况就会发生。“相连线程池”可能会引发一些问题，比如会导致`LinkedBlockingQueue`溢出。另外，如果消费者比生产者多，队列会因此而经常空着。好在Java 7在`BlockingQueue`上有了解决办法——`TransferQueue`。

5. `TransferQueue`——Java 7中的新贵

Java 7引入了`TransferQueue`。它本质上是多了一项`transfer()`操作的`BlockingQueue`。如果接收线程处于等待状态，该操作会马上把工作项传给它。否则就会阻塞直到取走工作项的线程出现。你可以把这看做“挂号信”选项，即正在处理工作项的线程在交付当前工作项之前不会开始其他工作项的处理工作。这样系统就可以调控上游线程池获取新工作项的速度。

用限定大小的阻塞队列也能达到这种调控效果，但TransferQueue接口更灵活。此外，用TransferQueue取代BlockingQueue的代码性能表现可能会更好。这是因为编写TransferQueue的实现时已经将现代编译器和处理器的特性考虑在内，执行起来效率更高。聊了这么久性能，不能空口无凭，必须给出测量结果并能证明才行。另外你也应该意识到，Java 7只给出了TransferQueue的一种实现形式——链表版。

在下面的例子中，你会发现用TransferQueue代替BlockingQueue是多么简单。只要对清单4-13中的代码做些简单修改，就可以升级成TransferQueue，请看这里。

代码清单4-14 用TransferQueue代替BlockingQueue

```
public abstract class MicroBlogExampleThread extends Thread {
  protected final TransferQueue<Update> updates;
  ...
public MicroBlogExampleThread(TransferQueue<Update> lbq_, int pause_) {
    updates = lbq_;
    pauseTime = pause_;
  }
  ...
  }
final TransferQueue<Update> lbq = new LinkedTransferQueue<Update>(100);
MicroBlogExampleThread t1 = new MicroBlogExampleThread(lbq, 10) {
  public void doAction(){
    ...
try {
      handed = updates.tryTransfer(u, 100, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
    }
    ...
    }
};
```

到此为止，用来开发多线程应用的主要构件我们都见识过了。接下来该把它们整合到驱动并发代码的引擎（执行器框架）上了。用它们可以对任务进行调度和控制，可以组合高效的并发流处理工作项，从而构建大型多线程应用程序。

4.4 控制执行

我们在前面的讨论中一直把工作任务当成抽象的单元。然而有个细节需要注意，我们一直没有提到的是这些单元要比Thread小——它们提供的方法把计算任务包含在一个工作单元中，无需为每个单元启动新的线程。这样处理多线程代码通常效率更高，因为免除了为每个单元启动Thread的开销。执行代码的线程是重用的，处理完一个任务后会继续处理新的工作单元。

虽然复杂一些，但你可以实现线程池、工人与管理者模式和执行者等开发人员最常用的模式。我们接下来要密切关注可以对任务（Callable、Future和FutureTask）和执行者建模的类和接口，特别是ScheduledThreadPoolExecutor。

4.4.1　任务建模

我们的终极目标是不用为调度每个任务或工作单元而启动新线程。归根结底，就是要把它们做成可以调用（通常由执行者调用）的代码，而不是直接可运行的线程。

我们来看对任务建模的三种办法——Callable和Future接口以及FutureTask类。

1. Callable接口

Callable接口是一个非常常见的概念，代表了一段可以调用并返回结果的代码。尽管这种做法很直接，但实际上它的作用微妙而又强大，用它可以创建出一些特别实用的模式。

Callable的典型用法是匿名实现类。这段代码的最后一行把s赋值为out.toString():

```
final MyObject out = getSampleObject();

Callable<String> cb = new Callable<String>() {
  public String call() throws Exception {
    return out.toString();
  }
};
String s = cb.call();
```

可以把Callable的匿名实现类当做对单一抽象方法call()的递延调用，该实现必须提供这个方法。

Callable是SAM类型（“单一抽象方法”的缩写，有时会这样称呼它）的示例——这是Java 7把函数作为一等类型最可行的办法。在后续章节讨论非Java语言时还会遇到它们，那时我们还会进一步讨论把函数作为值或一等类型的概念。

2. Future接口

Future接口用来表示异步任务，是还没有完成的任务给出的未来结果。我们在第2章介绍NIO.2和异步I/O时提过。

下面是Future中的主要方法。

- get()——用来获取结果。如果结果还没准备好，get()会被阻塞直到它能取得结果。还有一个可以设置超时的版本，这个版本永远不会阻塞。
- cancel()——在运算结束前取消。
- isDone()——调用者用它来判断运算是否结束。

下面这段代码（找素数）展示了Future的用法：

```
Future<Long> fut = getNthPrime(1_000_000_000);

Long result = null;
while (result == null) {
  try {
    result = fut.get(60, TimeUnit.SECONDS);
  } catch (TimeoutException tox) { }
  System.out.println("Still not found the billionth prime!");
}
System.out.println("Found it: "+ result.longValue());
```

在这段代码中，你应该想象一下返回Future的getNthPrime()在某个后台线程或多个线程上运行的情景，也有可能是在执行者框架上运行。即便使用先进的硬件，这种运算可能也需要很长时间——你最后还是要用Future的cancel()方法。

3. FutureTask类

FutureTask是Future接口的常用实现类，它也实现了Runnable接口。这意味着FutureTask可以由执行者调度，这一点很关键。它对外提供的方法基本上就是Future和Runnable接口的组合：get()、cancel()、isDone()、isCancelled()和run()，最后一个方法通常都是由执行者调用，你基本不需要直接调用它。

FutureTask还提供了两个很方便的构造器：一个以Callable为参数，另一个以Runnable为参数。这些类之间的关联表明对于任务建模的办法非常灵活，允许你基于FutureTask的Runnable特性（因为它实现了Runnable接口），把任务写成Callable，然后封装进一个由执行者调度并在必要时可以取消的FutureTask。

4.4.2 ScheduledThreadPoolExecutor

ScheduledThreadPoolExecutor（以下简称STPE）是线程池类中的重中之重——它功能多样，广受欢迎。STPE接收任务，并把它们安排给线程池里的线程。

- 线程池的大小可以预定义，也可自适应。
- 所安排的任务可以定期执行，也可只运行一次。
- STPE扩展了ThreadPoolExecutor类（很相似，但不具备定期调度能力）。

和java.util.concurrent中的工具类相结合的STPE线程池是大中型多线程应用程序最常见的模式之一，这些工具类包括我们在前面已经见过的ConcurrentHashMap、CopyOnWrite-ArrayList和BlockingQueue等。

STPE不过是通过Executors类的工厂方法轻易获取的众多执行者之一。使用这些工厂方法很方便，开发人员通过它们可以轻易获取典型配置，需要时还可以开放完整的接口方法。

下面的代码是一个定期读取的例子。这是newScheduledThreadPool()的常见用法：msgReader对象被安排poll()一个队列，从队列中的WorkUnit对象里取得工作项，然后输出。

代码清单4-15 STPE定期读取

```
private ScheduledExecutorService stpe;
private ScheduledFuture<?> hndl;                          <-- 取消时需要

private BlockingQueue<WorkUnit<String>> lbq = new LinkedBlockingQueue<>();

private void run(){
  stpe = Executors.newScheduledThreadPool(2);          <-- 执行者的工厂方法

  final Runnable msgReader = new Runnable(){
    public void run(){
      String nextMsg = lbq.poll().getWork();
      if (nextMsg != null) System.out.println("Msg recvd: "+ nextMsg);
```

```
      }
    };
    hndl = stpe.scheduleAtFixedRate(msgReader, 10, 10,
    ➥ TimeUnit.MILLISECONDS);
  }

  public void cancel() {
    final ScheduledFuture<?> myHndl = hndl;

    stpe.schedule(new Runnable() {
      public void run() { myHndl.cancel(true); }        <-- 取消时需要
    }, 10, TimeUnit.MILLISECONDS);
  }
```

在这个例子中，STPE每隔10毫秒就唤醒一个线程，让它尝试`poll()`一个队列。如果读取返回`null`（因为队列当前为空），则什么也不会发生，线程回去继续睡大觉。如果收到了一个工作单元，则线程会输出该工作单元的内容。

用Callable调用的代表性问题

形式简单的`Callable`、`FutureTask`及相关类存在几个问题——尤其在涉及类型系统时。

要明白这一点，可以想想怎么才能满足一个未知方法可能出现的所有方法签名。`Callable`只能用于没有参数的方法。要满足所有可能性，你需要`Callable`的不同变体。

在Java中，你可以通过指定模型系统内的方法签名来解决这个问题。但你在本书第三部分会见到，动态语言不能用这种静态视图来约束。我们将会返回来重点讨论这种类型系统之间的不匹配。现在你只要注意到，虽然`Callable`很有用，但要用它构建一个通用框架来对线程执行进行建模还是有点儿限制得太死了。

现在我们要转向Java 7重点突出的框架之一——用于轻量级并发的分支/合并（fork/join）框架。这个框架比我们在本节中见到的执行者在处理并发问题方面更加高效，要达到这点绝非易事。

4.5 分支/合并框架

就像第6章要讨论的一样，处理器的速度（或者更准确地说是CPU上的晶体管数量）最近几年增长迅猛。由此产生的副作用就是处理器等待I/O操作变成了家常便饭。这表明我们能够更好地利用计算机的处理能力。分支/合并框架就可以解决这个问题——这也是Java 7对并发领域新做出的最大贡献。

分支/合并框架完全是为了实现线程池中任务的自动调度，并且这种调度对用户来说是透明的。为了达到这种效果，必须按用户指定的方式对任务进行分解。在很多应用程序中，对于分支/合并框架来说都可以很自然地把其中的任务分成“小型”和“大型”任务。

我们来快速浏览一些与分支/合并框架相关的重要事实和基本原理。

- 引入了一种新的执行者服务，称为`ForkJoinPool`。
- `ForkJoinPool`服务处理一种比线程“更小”的并发单元`ForkJoinTask`。

- ForkJoinTask是一种由ForkJoinPool以更轻量化的方式所调度的抽象。
- 通常使用两种任务（尽管两种都表示为ForkJoinTask实例）：
  - “小型”任务是那种无需处理器耗费太多时间就可以直接执行的任务。
  - “大型”任务是那种需要在直接执行前进行分解（还可能不止一次）的任务。
- 提供了支持大型任务分解的基本方法，它还有自动调度和重新调度的能力。

这个框架的关键特性之一就是这些轻量的任务都能够生成新的ForkJoinTask实例，而这些实例将仍由执行它们父任务的线程池来安排调度。这就是分而治之。

我们会通过一个简单的例子告诉你如何使用分支/合并框架，然后简短介绍“工作窃取”这个特性，最后讨论一下那些适用于并行处理技术的特性。使用分支/合并框架最好的办法就是从例子入手。

4.5.1　一个简单的分支/合并例子

我们为说明分支/合并框架而设定了这样的应用场景：有一个数组，里面存放不同时间到达的微博更新，我们想按到达时间对它们排序，以便为用户生成时间线，就像在代码清单4-9中生成的那个一样。

我们会用MergeSort的变体实现多线程排序。代码清单4-16中用到了ForkJoinTask的特定子类RecursiveAction。因为它明显可以独立完成任务（对这些更新的排序能当即完成），而且具备递归处理能力（递归特别适合做排序），所以用RecursiveAction会比用通用的ForkJoinTask更简单。

MicroBlogUpdateSorter类用Update对象的compareTo()方法对更新列表排序。compute()方法（超类RecursiveAction中的抽象方法，必须实现）基本上是按创建时间对微博更新数组排序。

代码清单4-16　用RecursiveAction排序

```
public class MicroBlogUpdateSorter extends RecursiveAction {
  private static final int SMALL_ENOUGH = 32;        <-- 串行排序项只有32个或更少
  private final Update[] updates;
  private final int start, end;
  private final Update[] result;

  public MicroBlogUpdateSorter(Update[] updates_) {
    this(updates_, 0, updates_.length);
  }

  public MicroBlogUpdateSorter(Update[] upds_,
  ➥ int startPos_, int endPos_) {
    start = startPos_;
    end = endPos_;
    updates = upds_;
    result = new Update[updates.length];
  }

  private void merge(MicroBlogUpdateSorter left_,
```

```
➥ MicroBlogUpdateSorter right_) {
    int i = 0;
    int lCt = 0;
    int rCt = 0;
    while (lCt < left_.size() && rCt < right_.size()) {
      result[i++] = (left_.result[lCt].compareTo(right_.result[rCt]) < 0)
        ? left_.result[lCt++]
        : right_.result[rCt++];
    }
    while (lCt < left_.size()) result[i++] = left_.result[lCt++];
    while (rCt < right_.size()) result[i++] = right_.result[rCt++];
  }

  public int size() {
    return end - start;
  }

  public Update[] getResult() {
    return result;
  }

  @Override
  protected void compute() {                    ← RecursiveAction 中声明的方法
    if (size() < SMALL_ENOUGH) {
      System.arraycopy(updates, start, result, 0, size());
      Arrays.sort(result, 0, size());
    } else {
      int mid = size() / 2;
      MicroBlogUpdateSorter left = new MicroBlogUpdateSorter(
      ➥ updates, start, start + mid);
      MicroBlogUpdateSorter right = new MicroBlogUpdateSorter(
      ➥ updates, start + mid, end);
      invokeAll(left, right);
      merge(left, right)
    }
  }
}
```

要使用这个排序器，你可以用下面这样的代码驱动它，生成一些包含由X组成的字符串的更新，并打乱它们的顺序，之后再传给排序器。最终得到重新排序后的更新。

代码清单4-17　使用递归排序器

```
List<Update> lu = new ArrayList<Update>();
String text = "";
final Update.Builder ub = new Update.Builder();
final Author a = new Author("Tallulah");

for (int i=0; i<256; i++) {
  text = text + "X";
  long now = System.currentTimeMillis();
  lu.add(ub.author(a).updateText(text).createTime(now).build());
  try {
    Thread.sleep(1);
```

```
    } catch (InterruptedException e) {
    }
  }
  Collections.shuffle(lu);
  Update[] updates = lu.toArray(new Update[0]);

  MicroBlogUpdateSorter sorter = new MicroBlogUpdateSorter(updates);
  ForkJoinPool pool = new ForkJoinPool(4);
  pool.invoke(sorter);

  for (Update u: sorter.getResult()) {
    System.out.println(u);
  }
```

传入空数组，省掉空间分配

TimSort

随着Java 7的到来，默认的数组排序算法已经变了。以前是以QuickSort的形式，但到了Java 7时代则变成了“TimSort”——MergeSort和插入排序的混合体。TimSort最初是Tim Peters为Python开发的，而且从2.3版（2002）开始就是Python中的默认排序算法了。

如果想看看TimSort在Java 7中存在的证据，可以给清单4-16中的代码传入一个null数组。对数组排序时，由于数组尺寸太小，会调用Array.sort()方法，该方法会抛出空指针异常，在输出的异常信息里就能看到TimSort类。

4.5.2　ForkJoinTask与工作窃取

ForkJoinTask是RecursiveAction的超类。它是从动作中返回结果的泛型类型，所以RecursiveAction扩展了ForkJoinTask<Void>。这使得ForkJoinTask非常适合用MapReduce[①]方式返回数据集中归结出的结果。

ForkJoinTasks由ForkJoinPool调度安排，ForkJoinPool是专为这些轻量任务设计的新型执行者服务。该服务维护每个线程的任务列表，并且当某个任务完成时，它能把挂在满负荷线程上的任务重新安排到空闲线程上去。

采用这种“工作窃取”的算法是为了解决大小不同的任务所导致的调度问题。大小不同的任务所需的运行时间通常也会有很大差别。比如说，某个线程的运行队列中都是小任务，而另外一个全是大任务。如果小任务的运行速度比大任务快五倍，只处理小任务的线程很可能在处理大任务的线程完成之前就处于空闲状态了。

Java 7实现的工作窃取机制精准地解决了这个问题，并且在分支/合并框架工作的整个生命周期中使线程池中的所有线程都有用武之地。工作窃取完全是自动的，你什么也不用做就能享受到它带来的好处。不需要手工干预，而是由运行环境承担更多工作帮助开发人员管理并发，这在Java 7中已经不是什么新鲜事了。

① MapReduce是Google提出的一个软件架构，用于大规模数据集（大于1TB）的并行运算。当前的软件实现是指定一个Map（映射）函数，用来把一组键值对映射成一组新的键值对，指定并发的Reduce（化简）函数，用来保证所有映射的键值对中的每一个元素都共享相同的键组。——译者注

4.5.3　并行问题

分支/合并框架的确对我们的帮助很大，但在实际中，并不是每个问题都能像4.5.1节中那样轻易地简化成多线程MergeSort。

这里是一些可以用分支/合并方法解决的问题：

- 模拟大量简单对象的运动，比如粒子效果；
- 日志文件分析；
- 从输入中计数的数据操作，比如mapreduce操作。

从另一个角度来说，图4-10中这个被分解的问题正是分支/合并框架可以解决的。

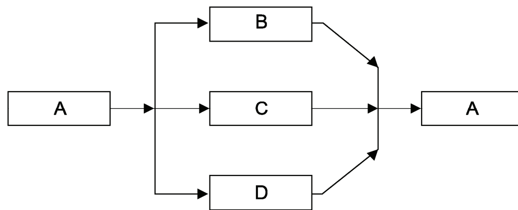


图4-10　分支与合并

用下面这个列表检查问题及其子任务是一个切实有效的方法，它可以确定是否能用分支/合并来解决这个问题。

- 问题的子任务是否无需与其他子任务有显式的协作或同步也可以工作?
- 子任务是不是不会对数据进行修改，只是经过计算得出些结果（它们是不是函数程序员称为“纯粹的”函数的函数）?
- 对于子任务来说，分而治之是不是很自然的事?子任务是不是会创建更多的子任务，而且它们要比派生出它们的任务粒度更细?

对于前面这些提问，如果答案是肯定的，或者“大体如此，但有临界情况”，那你的问题很可能适合用分支/合并的方式解决。反过来，如果答案是“可能吧”或者“算不上”，你就会发现分支/合并帮不上什么忙，可能用其他的同步方式更合适。

注意　前面的检查列表是测试某个问题（比如在Hadoop和NoSQL数据库中常见的那种）能否很好地用分支/合并方式解决的有效方法。

想设计出优秀的多线程算法并不容易，分支/合并方法也不能面面俱到。在适用的领域，它的用处很广。其实归根结底，你必须要确定你的问题是否适合这个框架，如果不适合，你只能在性能卓越的java.util.concurrent工具箱上构建自己的解决方案。

在下一节中，我们会详细讨论经常被误解的Java内存模型（Java Memory Model，JMM）。很多Java程序员都知道JMM，并且在没有经过正式介绍的情况下按自己的理解写代码。如果你觉得

这是在说你，那么接下来的内容会帮助你重新认识JMM，并且帮你打下扎实的基础。JMM这个话题相当有深度，所以如果你急着进入下一章，可以先跳过它。

4.6 Java 内存模型

Java语言规范（JLS）在第17.4节中介绍了JMM。其中的描述非常正式，用同步动作和被称为偏序[①]的数学结构描述JMM。这对于语言理论学家和Java规范的实现者（编译器和虚拟机的制造者）来说非常棒，但对于需要理解多线程代码如何执行的应用开发人员来说，这种描述会让他们头昏脑胀。

我们在这里不重复规范里的正式描述，而是用两个基本概念列出最重要的规则，这两个概念是代码块之间的之前发生（Happens-Before）和同步约束（Synchronizes-With）关系。

- 之前发生——这种关系表明一段代码块在其他代码开始之前就已经全部完成了。
- 同步约束——这意味着动作继续执行之前必须把它的对象视图与主内存进行同步。

如果你认真研究过OO编程，应该听到过面向对象构件的Has-A和Is-A这两种表述方式。一些开发人员发现，用之前发生和同步约束来描述基本的概念构件对理解Java并发很有帮助。这和Has-A与Is-A的道理一样，但这两组概念在技术上没有直接关联。

图4-11中是一个易失性写入与后续的读取访问（用于`println`）之间同步约束的例子。

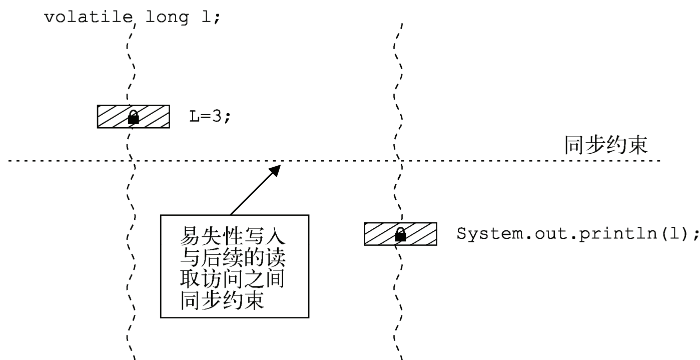


图4-11 同步约束的例子

JMM的主要规则如下。

- 在监测对象上的解锁操作与后续的锁操作之间存在同步约束关系。
- 对易失性（volatile）变量的写入与后续对该变量的读取之间存在同步约束关系。
- 如果动作A受到动作B的同步约束，则A在B之前发生。
- 如果在程序中的线程内A出现在B之前，则A在B之前发生。

① 设A是一个非空集，P是A上的一个关系，若关系P是自反的（对任意的a∈A，(a, a)∈P）、反对称的（若（a,b)∈P且（b，a)∈P，则a=b）和传递的（若（a,b)∈P，(b,c)∈P，则（a,c)∈P），则称P是集合A上的偏序关系。比如实数集上的小于等于关系（a<=a；a<=b，b<=a，则a=b；a<=b，b<=c，则a<=c）。——译者注

前两个规则通俗来说就是“先放后取”。换句话说，一个线程在写入时持有的锁要在其他操作（包括读取）能够获取锁之前被释放掉。

这里还有些规则，实际上是关于敏感行为的。

- 构造方法要在那个对象的终结器开始运行之前完成（一个对象被终结之前必须已经构造完整）。
- 开始一个线程的动作受到这个新线程的第一个动作的同步制约。
- `Thread.join()`受到被合并的线程的最后一个（和其他全部）动作的同步制约。
- 如果X在Y之前发生，并且Y在Z之前发生，则X在Z之前发生（传递性）。

这些简单的规则定义了内存和同步如何工作的全平台视图。图4-12展示了传递性规则。

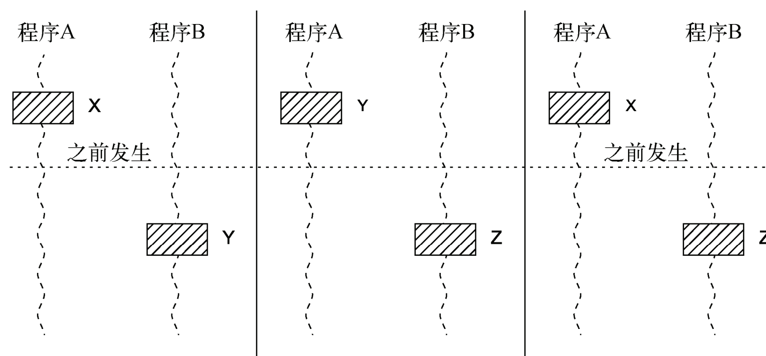


图4-12　之前发生的传递性

> **注意**　实际上，这些规则是JMM做出的最低保证。真正的JVM实际上可能表现得更好。对于开发人员来说，这可能是个陷阱，因为某个特定JVM中的行为实际上是个隐藏在底层并发中的诡异bug，却很容易给人造成错觉，以为是它提供的安全特性。

从这些最低保证中，很容易可以看出不可变性成为Java并发编程中的一个重要概念的原因。如果对象不可改变，确保改变对所有线程可见的相关问题就不会出现。

4.7　小结

并发是Java平台最重要的特性之一，扎实的并发编程知识对于一个优秀的开发人员来说日益重要。我们回顾了Java并发的基础和多线程系统的设计原则，并讨论了Java内存模型和Java平台如何实现并发的底层细节。

更重要的是我们解释了`java.util.concurrent`中的那些类和接口，现代Java开发人员在编写新的多线程代码时，更喜欢用到这些工具。我们还向你详细介绍了Java 7中一些新的类，如`LinkedTransferQueue`和分支/合并框架。

希望我们已经为你打好了基础，使你能够用java.util.concurrent中的类编写代码。这是本章内容的重中之重。尽管我们也探讨了一些核心理论，但最重要的还是实际样例。哪怕你刚开始使用ConcurrentHashMap和Atomic类，也能马上见识到这些经过严格测试的类所带来的好处。

时间到了，我们马上就要进入下一主题——一个能让你从Java开发者中脱颖而出的重要主题。在下一章，你会在Java平台的另一个基础领域（类加载和字节码）打下坚实的基础。这一领域是很多讨论平台安全和性能特性内容的核心，并巩固了生态系统内的很多先进技术。所以对于想鹤立鸡群的开发人员来说，这是个绝佳的研究课题。

第 5 章

类文件与字节码

本章内容

- 类加载
- 方法句柄
- 解剖类文件
- JVM字节码以及它的重要性
- 新的操作码invokedynamic

要成为优秀的Java开发人员，需要深入理解Java平台的工作方式。其中就包括类加载和JVM字节码这样的核心特性。

假设有一个大量使用依赖注入（DI）技术的应用程序，比如Spring，它在启动时出了问题，报了一个神秘的错误信息。如果不是简单的配置错误问题，你就需要了解如何实现DI框架才能跟踪问题来源。也就是说你得明白类加载机制。

或者假定跟你合作的开发商跑路了，只给你留下了一堆编译过的代码，没有源码，文档也乱七八糟的。你怎么能知道编译过的代码包含了什么呢？

最常见的程序启动失败错误就是ClassNotFoundException或NoClassDefFoundError，但很多开发人员都不知道它们是什么，有什么区别以及为什么会出现。

本章重点就是这些与开发相关的平台特性。此外还会讨论一些高级特性——它们是为Java的粉丝准备的，如果你时间有限，可以跳过那部分内容。

我们会从类加载的概览开始，这是VM为运行中的程序定位和激活新类型的过程。其核心是在VM中表示类型的Class对象。接下来我们会介绍一下新的方法句柄API，并和Java 6中已有的技术（比如反射）进行比较。

之后我们会讨论检查和分析类文件的工具。用Oracle JDK提供的javap作为参考工具。上过类文件的解剖课后，我们会转而讨论字节码，其中涉及JVM操作码的主要体系以及运行时的底层操作。

在你用字节码的知识把自己武装起来之后，我们会深入探讨invokedynamic操作码，它是Java 7新引入进来的，为的是让非Java语言能充分利用JVM的平台特性。

我们先从类加载开始吧，这是一个将新的类合并到正在运行着的JVM进程中的过程。

5.1 类加载和类对象

一个.class文件定义了JVM中的一个类型，包括域、方法、继承信息、注解和其他元数据。规范中对类文件的格式有详细描述，任何想在JVM上运行的语言都必须遵守。

类是平台能加载的最小程序代码单元。要将新的类加入到JVM的当前运行态中，有几步操作必须执行。首先，类文件必须被加载进来并连接，而且必须进行大量的验证。之后会提供一个代表该类型的新`Class`对象给正在运行的系统，并可以创建新的实例。

本节会讨论所有这些步骤，并介绍一下类加载器，也就是控制整个过程的那些类。我们先来看看加载和连接。

5.1.1 加载和连接概览

JVM的目的是使用类文件并执行其中的字节码。要实现这个目的，JVM必须以字节数据流的方式取出类文件中的内容，并将其转换成可用的格式加入运行态中。这个分两步走的过程被称为加载和连接，但连接又会被分解为几个子阶段。

加载

这个过程首先要读取构成类文件的字节数据流并给类的表现形式解冻。该过程一开始是创建一个字节数组，其内容通常是从文件系统中读取的，然后产生与所加载的类对应的`Class`对象。在这个过程中会对类做一些基本检查，但在加载过程结束时，`Class`对象还不成熟，所以类也不可用。

连接

加载完成之后，类必须被连接起来。这一步骤分为三个子阶段——验证，准备和解析。验证阶段证实类文件符合预期，不会引起系统的运行时错误或其他问题。之后是类的准备阶段，在类文件中引用的其他类型全部都要定位到，以确保该类已准备就绪。

连接步骤中各子阶段之间的相互关系如图5-1所示。

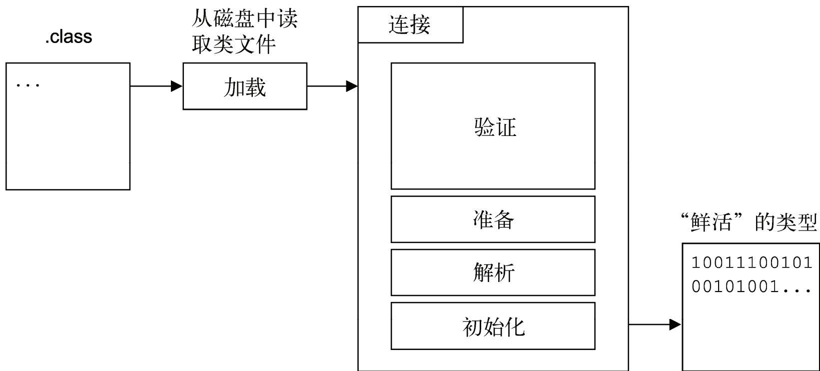


图5-1 加载与连接（及连接的子阶段）

5.1.2　验证

验证是一个非常复杂的过程，它分为几个步骤。

首先是完整性检查。这实际上是加载过程中的一部分，会确保类文件格式良好，可以连接。

接着是检查常量池（详情参见5.3.3节）中的符号信息是自相一致的，并要遵守常量的基本行为准则。其他不涉及代码的静态检查也要在这一阶段完成，比如检查`final`方法有没有被重写。

之后是验证中最复杂的部分——方法的字节码检查。要检查字节码行为良好，并且不会试图摆脱VM的环境控制。下面是一些主要检查。

- 是否所有方法都遵守访问控制关键字的限定。
- 方法调用的参数个数和静态类型是否正确。
- 确保字节码不会试图滥用堆栈。
- 确保变量使用之前被正确初始化了。
- 检查变量是否仅被赋予恰当类型的值。

做这些检查是出于性能方面的考虑，这样可以加快解释码的运行速度，运行时就不用再做这些检查了。同时还简化了运行时把字节码编译为机器码的过程（即时编译，详情参见6.6节）。

准备

类的准备包括分配内存和准备好初始化类中的静态变量，但不会现在初始化变量，也不会执行任何VM字节码。

解析

解析会促使VM检查类文件中所引用的类型是不是都是已知的类型。如果有运行时未知的类型，那它们也需要被加载进来。这些可见的未知类型会再次引发类加载过程。

一旦需要加载的其他类型全被定位并解析完成，VM就可以初始化那个最初要加载的类。这时所有静态变量都可以被初始化，所有静态初始化代码块都会运行。现在你运行的字节码就是来自新加载进来的类里的。这一步完成之后，类的加载就已全部完成，类也就可以使用了。

5.1.3　`Class`对象

连接和加载过程的最终结果是一个`Class`对象，用于表示加载并连接起来的新类型。尽管出于性能方面的考虑，`Class`对象只是在要求的地方做了初始化，但现在它在VM中完全生效了。代码可以继续执行了，它可以使用新类型并创建新实例。此外，`Class`对象提供了一些不错的方法，比如`getSuperClass()`，可以用它返回`Class`对象的父类。

`Class`对象可以和反射API一起实现对方法、域、构造方法等类成员的间接访问。`Class`对象中有对类成员`Method`和`Field`对象的引用。反射API可以用这些对象实现对它们的间接访问。图5-2是这种结构的高层视图。

运行时的哪个部分会定位并连接字节流以生成新的加载类？在下一个主题中，我们会讨论这个问题，即能够完成这些工作的类加载器，它是由抽象类`ClassLoader`的子类们组成的。

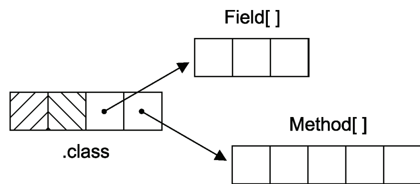


图5-2　Class对象与Method引用

5.1.4　类加载器

Java平台里有几个经典的类加载器，它们在平台的启动和常规操作过程中承担不同的任务：

- 根（或引导）类加载器——通常在VM启动后不久实例化，一般用本地代码实现。最好把它看做VM的一部分。它的作用通常是负责加载系统的基础JAR（主要是`rt.jar`），而且它不做验证工作。
- 扩展类加载器——用来加载安装时自带的标准扩展。一般包括安全性扩展。
- 应用（或系统）类加载器——这是应用最广泛的类加载器。它负责加载应用类。在大多数SE（Java标准版）的环境中，主要工作都是由它来完成。
- 定制类加载器——在更复杂的环境中，比如EE（Java企业版）或比较复杂的SE框架，通常会有些附加（即定制）的类加载器。有些团队甚至为他们的某个应用程序编写了特定的类加载器。

除了核心任务，类加载器还经常要从JAR文件或classpath中加载资源（不是类文件，比如图片或配置文件）。

例子——工具类加载器

在EMMA测试覆盖工具（http://emma.sourceforge.net/）中使用的一个类加载器可以作为加载时转化的例子。

当为了加上额外的测试辅助信息而加载类时，EMMA的类加载器会修改字节码。当在这些代码上运行测试用例时，EMMA会记录测试用例实际测试了哪些方法和代码分支。从这些记录中，开发人员能看出对一个类的单元测试是否全面。关于测试和覆盖，在11和12章还有更多的相关讨论。

有些框架和代码还经常会使用带有额外属性的专用（甚至用户自定义的）类加载器。这些类加载器经常会在加载时对字节码进行转换，我们在第1章有提到过。

图5-3中是类加载器的继承层级以及不同加载器之间的相互关系。

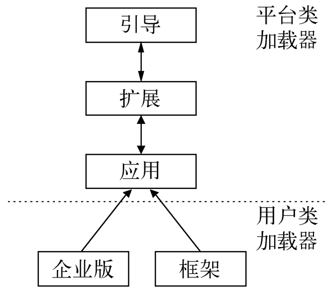


图5-3　类加载器层级

我们先来看一个专用类加载器的例子——如何用类加载实现依赖注入。

5.1.5　示例：依赖注入中的类加载器

DI有两个核心思想。

- 系统内的功能单元要靠依赖项和配置信息才能正常发挥作用。
- 在对象自身的上下文里，很难表示依赖项，即使可以，也很复杂难懂。

你脑海中应该浮现出一幅包含了行为，配置和依赖项信息（它们处在对象之外）的画面。这部分通常被称为对象的运行时路线。

第3章以Guice框架为例介绍了DI。本节中我们会讨论利用类加载器实现DI的方式，但这种方式与Guice不同，实际上它更像简化版的Spring框架。

在这个想象出来的DI框架下，我们像这样启动应用程序：

```
java -cp <CLASSPATH> org.framework.DIMain /path/to/config.xml
```

CLASSPATH中必须包含DI框架的JAR文件以及在config.xml文件中引用的所有类（以及它们的所有依赖项）的JAR文件。

我们改写了前面一个类似的例子——代码清单3-7中的服务，结果如清单5-1所示。

代码清单5-1　`HollywoodService`——不同的DI风格

```
public class HollywoodServiceDI {
  private AgentFinder finder = null;

  public HollywoodServiceDI() {}                 // 空的构造方法

  public void setFinder(AgentFinder finder) {
    this.finder = finder;                        // setter方法
  }

  public List<Agent> getFriendlyAgents() {      // 同代码清单3-7
    ...
  }

  public List<Agent> filterAgents(List<Agent> agents, String agentType) {
    ...                                          // 同代码清单3-7
  }
}
```

为了将它置于DI框架的管理之下，还需要一个配置文件：

```
<beans>
  <bean id="agentFinder" class="wgjd.ch03.WebServiceAgentFinder"
    ... />

  <bean id="hwService" class="wgjd.ch05.HollywoodServiceDI"
        p:finder-ref="agentFinder"/>
</beans>
```

在这种方式中，DI框架利用配置文件来确定要构造的对象。这个例子需要hwService和agentFinder两个bean，框架会为每个bean调用空构造方法，之后是setter方法（比如为HollywoodServiceDI的依赖项AgentFinder调用setFinder()）。

这说明类加载分为两个阶段。第一阶段由应用类加载器完成，负责加载DIMain及其引用的类。然后DIMain开始运行，并在main()的参数中得到配置文件的位置。

这时候，框架已经在JVM中运行起来了，但config.xml中指定的用户类还碰都没碰呢。实际上，在DIMain检查配置文件之前，框架不可能知道要加载什么类。

要启动config.xml中指定的应用配置，需要类加载的第二阶段。这要用到定制的类加载器。首先，要检查config.xml文件的一致性，确保它没有错误。然后，如果毫无差错，定制的类加载器就会试图从CLASSPATH中加载指定类型。如果任何一步失败了，整个过程就会被中止。

如果成功了，DI框架可以继续创建所需的实例，并调用实例上的setter方法。如果这些都顺利完成了，程序上下文就开始运行了。

我们简单介绍了一下Spring风格的DI方式，其中大量使用了类加载。在Java技术中，还有很多要用到类加载器及其相关技术的领域。下面是一些众所周知的例子：

- 插件架构；
- 厂商提供的或自主研发的框架；
- 从非正常位置（非文件系统或URL）获取类文件；
- Java EE；
- 任何需要在JVM进程已经启动后加入新的、未知代码的情况下。

我们对类加载的讨论就到此为止。让我们进入下一节，探讨Java 7为满足反射等需求而提供的新API。

5.2 使用方法句柄

如果你不熟悉Java的反射API（Class、Method、Field和它们的朋友），可以大致浏览一下（甚至跳过）这一节的内容。可如果你的代码库中有很多反射代码，那么你一定要认真读一读，因为它介绍了Java 7中取得相同效果的新办法，而且所用的代码更简洁。

Java 7为间接调用方法引入了新的API。其中的关键是java.lang.invoke包，即方法句柄。你可以把它看做反射的现代化方式，但它不像反射API那样有时会显得冗长、繁重和粗糙。

取代反射代码

反射中有很多套路化的代码。如果你写过一些反射代码，就不会忘记必须一次又一次地用Class[]指向内省方法的参数类型，并把该方法的参数都封装成Object[]，还要捕捉各种讨厌的异常以防出错，而且反射代码看起来也很不直观。

通过将反射代码转移到方法句柄，可以去掉套路化的代码，提高代码的可读性，这是大势所趋。

方法句柄是将invokedynamic（详情参见5.5节）引入JVM项目中的一部分。但其作用不仅限于invokedynamic的应用案例，在框架和常规用户代码中也有用武之地。接下来我们会先介绍方法句柄的基本技术；之后会给出一个例子与现有的各种方式进行比较，并总结出其中的差异。

5.2.1 MethodHandle

什么是MethodHandle？它是对可直接执行的方法（或域、构造方法等）的类型化引用，这是标准答案。还有一种说法：方法句柄是一个有能力安全调用方法的对象。

下面我们要获取一个带有两个参数的方法（但我们可能连这个方法的名字都不知道）的方法句柄，之后调用对象obj上的句柄，传入参数arg0和arg1：

```
MethodHandle mh = getTwoArgMH();

MyType ret;
try {
  ret = mh.invokeExact(obj, arg0, arg1);
} catch (Throwable e) {
    e.printStackTrace();
}
```

这种能力有些像反射，还有些像4.4节介绍的Callable接口。实际上，Callable是对方法调用能力建模的早期尝试。但它只适用于不带参数的方法。为了满足现实情况中不同参数组合和调用的可能，我们需要编写带有特定参数组合的其他接口。

Java 6中有很多这种代码，接口四处蔓延，让开发人员万分苦恼（比如耗光保存类信息的PermGen内存——见第6章）。相比较而言，方法句柄则适用于任何方法签名，不需要产生那么多小类。这要归功于新引入的MethodType类。

5.2.2 MethodType

MethodType是表示方法签名类型的不可变对象。每个方法句柄都有一个MethodType实例，用来指明方法的返回类型和参数类型。但它没有方法的名字和“接收者类型”，即调用的实例方法的类型。

用MethodType类中的工厂方法可以得到MethodType实例。这里有几个例子：

```
MethodType mtToString = MethodType.methodType(String.class);
MethodType mtSetter = MethodType.methodType(void.class, Object.class);
```

```
MethodType mtStringComparator = MethodType.methodType(int.class,
String.class, String.class);
```

这些MethodType实例分别表示toString()，setter方法（Object类的成员）和Comparator<String>定义的compareTo()方法的类型签名。MethodType实例一般都遵循相同的模式，第一个传入的参数是方法的返回类型，随后的参数是方法参数的类型（跟Class对象一样），如下所示：

```
MethodType.methodType(RetType.class, Arg0Type.class, Arg1Type.class, ...);
```

你看，现在可以用普通对象来表示不同的方法签名了，不需要再逐一为它们定义新类型。这也在最大程度上保证了类型安全性，而且办法还很简单。如果你想知道某个方法句柄能否用特定的参数集调用，可以检查该句柄的MethodType。

现在你应该明白MethodType是如何解决接口泛滥的问题了，接下来就去看看怎么得到指向类中方法的方法句柄吧。

5

5.2.3 查找方法句柄

下面的代码展示了如何得到指向当前类中toString()方法的方法句柄。注意，mtToString和toString()的签名完全一致，返回类型为String，没有参数。也就是说相应的MethodType实例是MethodType.methodType(String.class)。

代码清单5-2 查找方法句柄

```
public MethodHandle getToStringMH() {
  MethodHandle mh;
  MethodType mt = MethodType.methodType(String.class);
  MethodHandles.Lookup lk = MethodHandles.lookup();   <-- 获取上下文

  try {
     mh = lk.findVirtual(getClass(), "toString", mt);   <-- 从上下文中查找方法句柄
  } catch (NoSuchMethodException | IllegalAccessException mhx) {
    throw (AssertionError)new AssertionError().initCause(mhx);
  }
  return mh;
}
```

取得新的方法句柄要用lookup对象，比如代码清单5-2中的lk。这个对象可以提供其所在环境中任何可见方法的方法句柄。

要从lookup对象中得到方法句柄，你需要给出持有所需方法的类、方法的名称，以及跟你所需的方法签名相匹配的 MethodType。

注意 在查找上下文（lookup context）中可以得到任何类型（包括系统类型）中的方法句柄。当然，如果要从没有关联的类中取得句柄，查找上下文中只能看到或取得public方法的句柄。就是说方法句柄总是在安全管理之下安全使用——没有反射中setAccessible()那种破解方法。

现在你已经拿到了方法句柄，接下来自然是执行它。方法句柄API为此提供了两个方法：invokeExact()和invoke()。invokeExact()方法要求其参数类型与底层方法所期望的参数类型完全匹配。invoke()方法会在参数类型不太正确时做些修改，以使其与底层方法参数相匹配（比如在需要时进行装箱或拆箱）。

接下来我们会给出一个长一点儿的例子，说明如何使用方法句柄取代过去的技术，比如反射和小型代理类。

5.2.4　示例：反射、代理与方法句柄

如果你曾经处理过满是反射的代码库，就会深知反射代码所带来的痛苦了。在本节中，我们要向你证明方法句柄可以取代很多套路化的反射代码，会让你的编码生涯更轻松。

代码清单5-3是改编自前面章节的例子。ThreadPoolManager负责将新任务分配给线程池，和代码清单4-15稍有不同。它还能取消正在运行的任务，但是个私有方法。

为了阐明方法句柄和其他技术之间的差别，我们给出了从外部访问类的私有方法cancel()的三种办法：makeReflective、makeProxy和makeMh。我们还展示了两种Java 6技术：反射和代理类。并且和基于MethodHandle的方式进行了比较。我们用到了一个读取队列的任务QueueReaderTask（实现了Runnable接口）。你可以在本章源码中找到QueueReaderTask实现。

代码清单5-3　三种访问方式

```
public class ThreadPoolManager {

  private final ScheduledExecutorService stpe =
Executors.newScheduledThreadPool(2);
  private final BlockingQueue<WorkUnit<String>> lbq;

  public ThreadPoolManager(BlockingQueue<WorkUnit<String>> lbq_) {
    lbq = lbq_;
  }

  public ScheduledFuture<?> run(QueueReaderTask msgReader) {
    msgReader.setQueue(lbq);
    return stpe.scheduleAtFixedRate(msgReader, 10, 10,
TimeUnit.MILLISECONDS);
  }

  private void cancel(final ScheduledFuture<?> hndl) {       <-- 要访问的私有方法
    stpe.schedule(new Runnable() {
      public void run() { hndl.cancel(true); }
    }, 10, TimeUnit.MILLISECONDS);
  }

  public Method makeReflective() {
    Method meth = null;

    try {
      Class<?>[] argTypes = new Class[] { ScheduledFuture.class };
      meth = ThreadPoolManager.class.getDeclaredMethod("cancel",
argTypes);
```

```
      meth.setAccessible(true);
    } catch (IllegalArgumentException | NoSuchMethodException
  | SecurityException e) {
      e.printStackTrace();
    }

    return meth;
  }

  public static class CancelProxy {
    private CancelProxy() { }

    public void invoke(ThreadPoolManager mae_, ScheduledFuture<?> hndl_) {
      mae_.cancel(hndl_);
    }
  }

  public CancelProxy makeProxy() {
    return new CancelProxy();
  }

  public MethodHandle makeMh() {
    MethodHandle mh;
    MethodType desc = MethodType.methodType(void.class,
  ScheduledFuture.class);

    try {
      mh = MethodHandles.lookup()
  .findVirtual(ThreadPoolManager.class, "cancel", desc);

    } catch (NoSuchMethodException | IllegalAccessException e) {
      throw (AssertionError)new AssertionError().initCause(e);
    }

    return mh;
  }
}
```

要求访问私有方法

创建**MethodType**

查找**MethodHandle**

这个类提供了三个访问私有方法`cancel()`的方法。实际上，一般实现时只会用一个，我们是为了讨论它们之间的差别才全都列了出来。

下面是使用这些方法的例子。

代码清单5-4　使用这些访问方法

```
private void cancelUsingReflection(ScheduledFuture<?> hndl) {
  Method meth = manager.makeReflective();

  try {
    System.out.println("With Reflection");
    meth.invoke(hndl);
  } catch (IllegalAccessException | IllegalArgumentException
| InvocationTargetException e) {
    e.printStackTrace();
  }
}
```

```
private void cancelUsingProxy(ScheduledFuture<?> hndl) {
  CancelProxy proxy = manager.makeProxy();

  System.out.println("With Proxy");
  proxy.invoke(manager, hndl);            <-- 通过代理调用是静态类型的
}

private void cancelUsingMH(ScheduledFuture<?> hndl) {
  MethodHandle mh = manager.makeMh();

  try {
    System.out.println("With Method Handle");
    mh.invokeExact(manager, hndl);        <-- 方法签名必须完全一致
  } catch (Throwable e) {                 <-- 必须捕捉Throwable
    e.printStackTrace();
  }
}

BlockingQueue<WorkUnit<String>> lbq = new LinkedBlockingQueue<>();
manager = new ThreadPoolManager(lbq);

final QueueReaderTask msgReader = new QueueReaderTask(100) {
  @Override
  public void doAction(String msg_) {
    if (msg_ != null) System.out.println("Msg recvd: "+ msg_);
  }
};
hndl = manager.run(msgReader);            <-- 然后用hndl取消任务
```

这几个cancelUsing方法都有一个ScheduledFuture参数，所以你可以用前面的代码试验不同的取消方法。实际上，作为API的使用者，你可以不用去管这是如何实现的。

在下一节中，我们会告诉你API或框架开发人员用方法句柄取代其他方式的原因。

5.2.5　为什么选择MethodHandle

在上一节中我们看了一个把方法句柄用在Java 6中使用反射和代理的地方的例子。这引出了一个问题：为什么要用方法句柄取代过去的处理方式？

从表5-1可以看出，反射最大的优势就是人们熟悉它。代理对于简单用例可能更容易理解，但我们认为方法句柄在这两方面做得都是最棒的。我们强烈推荐你使用方法句柄。

表5-1　Java的方法间接访问技术比较

特　　性	反　　射	代　　理	方法句柄
访问控制	必须使用setAccesible()。会被安全管理器禁止	内部类可以访问受限方法	在恰当的上下文中对所有方法都有完整的访问权限。和安全管理器没有冲突
类型纪律（Type discipline）	没有。不匹配就抛出异常	静态的。过于严格。为了存储全部的代理类，可能需要很多PermGen	在运行时是类型安全的。不占用PermGen
性能	跟其他的比算慢的	跟其他方法调用一样快	力求跟其他方法调用一样快

方法句柄还有一个特性，可以从静态上下文中确定当前类。如果你曾经编写过这样的日志代码（比如log4j）：

```
Logger lgr = LoggerFactory.getLogger(MyClass.class);
```

你应该知道这样的代码很脆弱。如果它被重构进超类或子类中，显式声明的类名就会有问题。然而在Java 7中，你可以这样写：

```
Logger lgr = LoggerFactory.getLogger(MethodHandles.lookup().lookupClass());
```

在这行代码中，可以把`lookupClass()`看成用在静态上下文中的`getClass()`。这在处理日志框架之类的场合中特别有用，因为通常每个用例都有自己的logger。

带着新掌握的方法句柄技术，我们去检查一下类文件的底层细节和使其变得有意义的工具。

5.3 检查类文件

类文件是二进制块，所以想直接和它打交道不太容易。但有很多时候你会发现必须和类文件交手。

比如说，为了在运行时更好地监控（比如通过JMX）应用程序，你需要加上额外的公共方法。重新编译和再次部署看起来顺利完成了，但检查管理API时却发现没有那些方法。又进行了几次构建和部署还是没有发现。

为了找出部署问题，你需要检查一下`javac`产生的类文件是不是你想要的那个。还有时候你需要研究那些没有源码的类文件，以验证文档中是不是真有你所怀疑的错误。

对于类似的任务，你必须用工具检查类文件的内容。好在标准的Oracle JVM中有`javap`这个工具，用它来探视类文件内部和反汇编类文件非常得心应手。

我们一开始会先介绍`javap`，以及为检查类文件而设置的各种基本参数。接下来会讨论方法名称和类型在JVM内部的一些表示方式。然后看一下常量池，它是JVM的“藏宝箱”，对于理解字节码如何工作非常重要。

5.3.1 介绍javap

`javap`的用处很多，既能看类声明了什么方法，又能输出字节码。我们来看一下`javap`最简单的用途，在第4章讨论的微博`Update`上试一下。

```
$ javap wgjd/ch04/Update.class
Compiled from "Update.java"
public class wgjd.ch04.Update extends java.lang.Object {
  public wgjd.ch04.Author getAuthor();
  public java.lang.String getUpdateText();
  public int hashCode();
  public boolean equals(java.lang.Object);
  public java.lang.String toString();
  wgjd.ch04.Update(wgjd.ch04.Update$Builder, wgjd.ch04.Update);
}
```

默认情况下，`javap`会显示访问权限为`public`、`protected`和默认（即包级`protected`）级别的方法。加上`-p`选项后还可以显示`private`方法和域。

5.3.2　方法签名的内部形式

JVM内部用的方法签名和javap显示出来供人阅读的形式不太一样。随着我们对JVM的不断深入，这些内部名称出现将更加频繁。如果你赶时间，可以跳过这一节。但请记住它，因为你可能还要回来参考这些内容。

在紧凑形式中，类型名称是经过压缩的。比如int是用I表示的。这些紧凑形式有时被称为类型描述符。表5-2中是类型描述符的完整列表。

表5-2　类型描述符

描　述　符	类　　型
B	byte
C	char（16位Unicode字符）
D	double
F	float
I	int
J	Long
L<类型名称>	引用类型（比如Ljava/lang/String; 用于字符串）
S	short
Z	boolean
[	array-of

某些情况下，类型描述符可能比类型名称还要长（比如Ljava/lang/Object就比Object长），但类型描述符是完全限定的，所以可以直接解析。

javap还有一个有用的选项-s，可以输出签名的类型描述符，所以你没必要用那个表自己做转换。你可以使用javap高级一些的方法来显示我们之前看过的一些方法的签名：

```
$ javap -s wgjd/ch04/Update.class
Compiled from "Update.java"
public class wgjd.ch04.Update extends java.lang.Object {
  public wgjd.ch04.Author getAuthor();
    Signature: ()Lwgjd/ch04/Author;

  public java.lang.String getUpdateText();
    Signature: ()Ljava/lang/String;

  public int compareTo(wgjd.ch04.Update);
    Signature: (Lwgjd/ch04/Update;)I

  public int hashCode();
    Signature: ()I

  ...
}
```

如你所见，方法签名中的所有类型都是用类型描述符表示的。

在下一节中你会看到类型描述符的另一个用途。它会出现在类文件中非常重要的部分——常量池。

5.3.3 常量池

常量池是为类文件中的其他（常量）元素提供快捷访问方式的区域。如果你研究过C或Perl之类的语言，应该知道符号表，对于JVM来说，常量池就类似于符号表。但和其他语言不同，Java没有完全开放对常量池中信息的访问。

为了不纠缠于过多的细节，我们用一个非常简单的例子来演示常量池。下面是一个简单的“游戏围栏”或者叫“演算本”类。我们在这个类的`run()`里面写一点代码，就可以快速测试Java的语法特性或类库。

代码清单5-5 游戏围栏样例类

```
package wgjd.ch04;

public class ScratchImpl {
  private static ScratchImpl inst = null;

  private ScratchImpl() {
  }

  private void run() {
  }

  public static void main(String[] args) {
    inst = new ScratchImpl();
    inst.run();
  }
}
```

要查看常量池中的信息，可以用`javap-v`。这个命令还会输出很多其他信息，不过我们只关注常量池中的条目。

如下所示：

```
 #1 = Class         #2        //  wgjd/ch04/ScratchImpl
 #2 = Utf8          wgjd/ch04/ScratchImpl
 #3 = Class         #4        //  java/lang/Object
 #4 = Utf8          java/lang/Object
 #5 = Utf8          inst
 #6 = Utf8          Lwgjd/ch04/ScratchImpl;
 #7 = Utf8          <clinit>
 #8 = Utf8          ()V
 #9 = Utf8          Code
#10 = Fieldref      #1.#11
➥ // wgjd/ch04/ScratchImpl.inst:Lwgjd/ch04/ScratchImpl;
#11 = NameAndType   #5:#6     //  instance:Lwgjd/ch04/ScratchImpl;
#12 = Utf8          LineNumberTable
#13 = Utf8          LocalVariableTable
#14 = Utf8          <init>
#15 = Methodref     #3.#16    //  java/lang/Object."<init>":()V
#16 = NameAndType   #14:#8    //  "<init>":()V
#17 = Utf8          this
#18 = Utf8          run
#19 = Utf8          ([Ljava/lang/String;)V
#20 = Methodref     #1.#21    //  wgjd/ch04/ScratchImpl.run:()V
```

```
#21 = NameAndType  #18:#8    //  run:()V
#22 = Utf8         args
#23 = Utf8         [Ljava/lang/String;
#24 = Utf8         main
#25 = Methodref    #1.#16    //  wgjd/ch04/ScratchImpl."<init>":()V
#26 = Methodref    #1.#27
   //  wgjd/ch04/ScratchImpl.run:([Ljava/lang/String;)V
#27 = NameAndType  #18:#19   //  run:([Ljava/lang/String;)V
#28 = Utf8         SourceFile
#29 = Utf8         ScratchImpl.java
➡ //  wgjd/ch04/ScratchImpl.run:([Ljava/lang/String;)V
#27 = NameAndType  #18:#19   //  run:([Ljava/lang/String;)V
#28 = Utf8         SourceFile
#29 = Utf8         ScratchImpl.java
```

如你所见，常量池中的条目是带有类型的。它们还会相互引用，比如说，一个类型为Class的条目会引用类型为Utf8的条目。而Utf8的条目是个字符串，所以Class条目引用的Utf8条目应该是类的名称。

表5-3是可能出现在常量池中的条目集。在讨论常量池中的条目时，有时会用CONSTANT_前缀，比如CONSTANT_Class。

表5-3 常量池条目

名 称	描 述
Class	类常量。引用类的名称（Utf8 条目）
Fieldref	定义域。引用该域的Class 和 NameAndType
Methodref	定义方法。引用该方法的Class和NameAndType
InterfaceMethodref	定义接口方法。引用该方法的Class 和 NameAndType
String	字符串常量。引用保存字符的Utf8常量
Integer	整型常量（4字节）
Float	浮点常量（4字节）
Long	长整型常量（8字节）
Double	双精度浮点型常量（8字节）
NameAndType	描述名称和类型对。类型引用一个保存类型描述符的Utf8条目
Utf8	一个表示以Utf8编码的字符的二进制字节流
InvokeDynamic	（Java 7中新引入的）见5.5节
MethodHandle	（Java 7中新引入的）描述MethodHandle常量
MethodType	（Java 7中新引入的）描述MethodType常量

你可以用这个表格从演算类的常量池中看到常量解析的例子。比如条目#10中的Fieldref。

要解析一个域，你需要名称、类型，还有它所在的类：#10的值是#1.#11，这就是说常量#11来自类#1。在输出中可以很容易看出#1确实是一个Class类型的常量，并且#11是NameAndType。#1指向ScratchImpl类本身，#11是#5:#6——一个名称为inst的ScratchImpl变量。所以综合来看，#10指向ScratchImpl类内部的自身静态变量inst（你可能已经从清单5-6的输出中猜出来了）。

在类加载过程中的验证环节，有一步是检查类文件中的静态信息是否一致的。前面的例子是运行时在加载新类时要做的完整性检查。

对于类文件的基本结构，我们已经讨论的差不多了。接下来要进入下一话题——字节码。理解源码如何变成字节码会对你理解代码如何运行有很大的帮助。在学习第6章以及后面的章节时，还能引导你更加深入地了解平台的能力。

5.4 字节码

到目前为止，在我们的讨论中，字节码一直有点幕后工作者的意思。我们先来回顾一下对它已经有了哪些了解，然后再对它进行详细介绍：

- 字节码是程序的中间表示形式：介于人类可读的源码和机器码之间。
- 字节码是通过javac处理源码文件产生的。
- 某些高层语言特性在编译时已经从字节码中去掉了。比如说Java的循环结构（for、while等）在字节码中就被转换成了分支指令。
- 每个操作码都由一个字节表示（因此被叫做*字节码*）。
- 字节码是一种抽象表示法，不是“某种虚拟CPU的机器码”。
- 字节码可以进一步编译成机器码，通常是“即时编译”。

字节码解释起来有点像先有鸡还是先有蛋的问题。要彻底搞清楚状况，你既要懂字节码，又要明白执行它的运行时环境。

这是一个循环依赖，为了解决这个问题，我们先来探索一个相对简单的例子。即使这一次你不太明白，也可以在后续章节读到更多字节码相关内容时再回来看看。

在例子之后，我们会给出一些与运行时环境相关的上下文和JVM操作码的目录（其中包括用于数学计算、调用、快捷形式之类的字节码）。最后，我们会用另外一个基于字符串拼接的例子来结束。现在就先去看看如何检查.class文件的字节码吧。

5.4.1 示例：反编译类

用带有-c选项的javap可以对类进行反编译。我们会以代码清单5-5中的演算类为例，主要检查方法之内的字节码。我们还会加上-p选项，以便能见到私有方法内的字节码。

我们一节一节的来——javap输出的每一部分都有很多信息，很容易让人不堪重负。首先，让我们先看头部。这里没什么特别出人意料或让人喜出望外的：

```
$ javap -c -p wgjd/ch04/ScratchImpl.class
Compiled from "ScratchImpl.java"
public class wgjd.ch04.ScratchImpl extends java.lang.Object {
  private static wgjd.ch04.ScratchImpl inst;
```

接下来是静态块。变量的初始化就放在这里，所以这表示inst被初始化为null了。看起来putstatic可能是一个把值放到静态域中的字节码。

```
static {};
  Code:
     0: aconst_null
     1: putstatic     #10   // Field inst:Lwgjd/ch04/ScratchImpl;
     4: return
```

代码前面的数字表示从方法开始算起的字节码偏移量。所以字节1是putstatic操作码，字节2和3表示一个16位的常量池索引，这个16位索引在这里的值是10，表示该值（此处为null）会存在常量池的条目#10所指明的域中。从字节码流开始的第4个字节是return操作符，表明这个代码块结束了。

接下来是构造方法。

```
private wgjd.ch04.ScratchImpl();
  Code:
     0: aload_0
     1: invokespecial #15   // Method java/lang/Object."<init>":()V
     4: return
```

在Java中，void构造方法总会隐式调用超类中的构造方法。这从上面的字节码里就能看出来invokespecial指令。一般来说，任何方法调用都会转换成VM的某一调用指令。

在run()方法中没有代码，因为这只是一个空白的演算类。

```
private void run();
  Code:
     0: return
```

在main方法中，你初始化了inst，还做了点对象创建。这说明了辨识通用字节码的基本模式：

```
public static void main(java.lang.String[]);
  Code:
     0: new           #1    // class wgjd/ch04/ScratchImpl
     3: dup
     4: invokespecial #21   // Method "<init>":()V
```

这种3个字节码指令的模式——new、dup和一个<init>的invokespecial——都表示创建新实例。

操作码new只为新实例分配内存。dup复制栈顶上的元素。要完整创建该对象，你需要调用构造方法的代码块。<init>方法中包含构造方法的代码，所以可以用invokespecial调用那段代码。我们继续看main方法中其余的字节码：

```
      7: putstatic     #10   // Field inst:Lwgjd/ch04/ScratchImpl;
     10: getstatic     #10   // Field inst:Lwgjd/ch04/ScratchImpl;
     13: invokespecial #22   // Method run:()V
     16: return
}
```

指令7保存刚刚创建的单例实例。指令10把它放回到栈顶上，这样指令13就可以调用它上面的方法了。注意，因为调用的run()是私有方法，所以13是invokespecial。私有方法不能重写，所以不能用Java的标准虚拟查询。大多数方法调用都会转换成invokevirtual指令。

注意　通常来说，javac产生的字节码没有经过特别优化，是非常简单的表示形式。基本策略是由JIT编译器来完成大部分的优化工作，所以简单直白的起点对它们是很有帮助的。VM实现者表示，“字节码就应该傻傻的”，这是他们对从源语言产生的字节码的总体感觉。

接下来我们要讨论字节码所需的运行时环境，之后会介绍用来描述字节码指令主要“家庭成员”的表格，其中包括加载/存储，数学计算，执行控制，方法调用和平台操作。然后我们会讨论操作码可能的快捷形式，最后会再给出一个例子。

5.4.2　运行时环境

因为JVM使用堆栈机，所以理解堆栈机的操作对理解字节码至关重要。

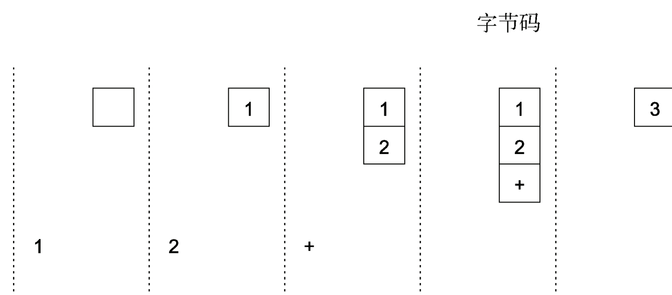


图5-4　将栈用于数学运算

JVM与硬件CPU（比如x64或ARM芯片）最显著的差别在于它没有处理器寄存器，而是用栈完成所有的计算和操作。有时候也这也被称为操作数栈（或计算堆栈）。图5-4展示了如何用操作数栈完成两个int数值的相加运算。

正如我们前面讨论过的，当一个类被链接进运行时环境时，它的字节码会受到检查，并且其中很多验证都可以归结为对栈中类型模式的分析。

注意　栈中的值只有类型正确时对它的处理才能生效。比如，如果我们把对一个对象的引用压入栈，然后试图将其作为int型进行数学计算，就可能会发生未定义或糟糕的事情。类加载过程中的验证阶段会进行广泛的检查，以确保新加载的类中不会有滥用栈的方法。这样做能够防止系统接受了损坏（或恶意）的类并引发问题。

方法在运行时需要一块内存区域作为计算堆栈来计算新值。另外，每个运行的线程都需要一个调用堆栈（栈跟踪中会报告的那个栈）来记录当前正在执行的方法。在某些情况下，这两个栈会有交互。看下面这行代码：

```
return 3 + petRecords.getNumberOfPets("Ben");
```

要计算出这行代码的结果，需要把3压入操作数栈。然后调用方法计算Ben有多少只宠物。为此，你需要把接收对象（方法属主，即petRecords）压入计算堆栈，要传入的所有参数尾随其后。

然后invoke操作符会调用方法getNumberOfPets()，把控制权移交给被调用的方法，刚刚进入的方法会出现在调用堆栈中。但进入新方法后，需要启用不同的操作数栈，所以已经在调用者的操作数栈中的值不可能影响被调用方法的计算结果。

在getNumberOfPets()完成时，返回结果会被放到调用者的操作数栈中，进程中与getNumberOfPets()相关的部分也会从调用堆栈中移走。然后相加运算可以得到两个值并把它们加在一起。

现在我们开始审视字节码。这是个大课题，而且有很多特殊情况，所以我们即将呈现的只是主要特性的概览，而不是完整的介绍。

5.4.3　操作码介绍

JVM字节码由操作码（opcode）序列构成，每个指令后面可能会跟着一些参数。操作码希望看到栈处于指定状态中，然后它对栈进行转换，把参数移走，放入结果。

每个操作码都由一个单字节值表示，所有最多只能有255个操作码。当前仅用了200个左右。对我们来说，把它们全列出来有点儿太多了，好在大多数操作码都可以归为几大族系。我们会逐一对这些族系进行讨论，帮助你理解它们。还有一些操作码不好界定应该归为哪一族系，但好在你不会经常遇见它们。

注意　JVM不是纯粹的面向对象运行时环境——它支持原始类型。这在某些操作码族系中有所体现——其中一些基本操作码类型（比如存储和相加）要有一些变体，在处理原始类型时会有所不同。

操作码表有四列：

- 名称：这是操作码类型的通用名称。大多数情况下，都会有几个相关的操作码在做类似的事情。
- 参数：操作码的参数。以i打头的参数是用来作为常量池或局部变量中的查询索引的几个字节。如果有更多的此类参数，它们会合并在一起，所以i1，i2表示“从这两个字节中生成一个16位的索引”。如果参数出现在括号里，就表明不是所有形式的操作码都会使用它。
- 堆栈布局：它展示了栈在操作码执行前后的状态。括号中的元素表明不是所有形式的操作码都使用它们，或者这些元素是可选的（比如调用操作码）。
- 描述：操作码的用处。

我们从表5-4中拿过来一行代码做例子，检查一下操作码getfield的条目。这个操作码用于从对象的域中读出一个值。

```
getfield i1, i2 [obj] → [val]        从栈顶端对象的常量池中取出指定位置的域。
```

第一列给出了操作码的名字：getfield。后面一列说明在字节码流中有两个参数跟在操作码后面。这些参数合在一起构成一个16位的值，可以用来从常量池里找到想要的域（记住常量池的索引总是16位的）。

堆栈布局那一列表明在找到栈顶端对象的类的常量池中的索引位置之后，该对象被移除，它的位置被那个域的值所替代。

这种把移走对象作为操作一部分的模式是一种让字节码变得紧凑的办法，没有繁琐的清理工作，也不用记着要挪走处理完的对象实例。

5.4.4 加载和储存操作码

加载和储存操作码这个族系负责将值加载到栈或检索值。表5-4给出了加载/储存族系的主要操作。

表5-4 加载和储存操作码

名 称	参 数	堆栈布局	描 述
load	(i1)	[] • [val]	从局部变量加载值（原始型或引用型）到栈上。有快捷形式，并且有针对不同类型的变体
ldc	i1	[] • [val]	从池中加载常量到栈上，针对不同类型有不同的变体，并且范围广泛
store	(i1)	[val] • []	把值（原始型或引用型）从进程的栈中移走，存到局部变量中。有快捷形式，有针对不同类型的变体
dup		[val] • [val, val]	复制栈顶部的值，有不同形式的变体
getfield	i1, i2	[obj] • [val]	从栈顶部对象的常量池中得到指定位置的域
putfield	i1, i2	[obj,val] • []	把值放入对象在常量池中指定位置的域上

前面提过，加载和储存指令有很多不同形式的变体。比如用来把双精度数从局部变量加载到栈上的dload操作码，以及用来把对象引用从栈弹出到局部变量中的astore操作码。

5.4.5 数学运算操作码

这些操作符在栈上执行数学运算。它们从栈顶端取出参数并进行计算。这些参数（总是原始型）必须完全匹配，但平台提供了很多对原始型进行类型转换的操作码。表5-5给出了基本的数学运算操作码。

类型转换（cast）操作码的名称非常短，比如i2d是把int转为double的操作码。需要特别说明的是，类型转换操作码中并没有cast，所以在表5-5中用括号把它括了起来。

表5-5 数学运算操作码

名 称	参 数	堆栈布局	描 述
add		[val1, val2] • [res]	把栈顶端的两个值相加（必须是相同的原始类型），并把结果存在栈中。有快捷形式，有针对不同类型的变体
sub		[val1, val2] • [res]	把栈顶端的两个值相减（必须是相同的原始类型），并把结果存在栈中。有快捷形式，有针对不同类型的变体

（续）

名称参数	堆栈布局	描述
div	[val1, val2] • [res]	把栈顶端的两个值相除（必须是相同的原始类型），并把结果存在栈中。有快捷形式，有针对不同类型的变体
mul	[val1, val2] • [res]	把栈顶端的两个值相乘（必须是相同的原始类型），并把结果存在栈中。有快捷形式，有针对不同类型的变体
(cast)	[value] • [res]	把值从一种原始类型转换为另外一种。每一种可能的类型转换都有对应的形式

5.4.6 执行控制操作码

如前所述，高级语言的控制结构在JVM字节码中没有出现。相反，流程控制是由很少的几个原始指令完成的，如表5-6所示。

表5-6 流程控制操作码

名称	参数	堆栈布局	描述
if	b1, b2	[val1, val2] • []或[val1] • []	如果符合特定条件，则跳转到特定分支的偏移处
goto	b1, b2	[] • []	无条件地跳转到分支偏移处。有宽大形式
jsr	b1, b2	[] • [ret]	跳到本地子流程中，并把返回地址（下一个操作码的偏移地址）放到栈中。有宽大形式
ret	索引	[] • []	返回到索引的局部变量所指向的偏移地址
tableswitch	{依情况而定}	[index] • []	用于实现switch
lookupswitch	{依情况而定}	[key] • []	用于实现switch

就像用于查找常量的索引字节，参数b1、b2用于构造方法内部的字节码跳转地址。jsr指令用于访问主流程之外一个自成体系的字节码区域（偏移地址可能在方法的主字节码之外）。在某些情况下，比如在异常处理块中，可能会用到它。

goto和jsr指令的宽大形式要用4个字节的参数，并且所构造的偏移量大于64 KB。但这并不常用。

5.4.7 调用操作码

调用操作码中有四个操作码可以处理普通的方法调用，还有一个Java 7中新出的特别操作码invokedynamic（5.5节有更多细节）。这五个方法调用操作码如表5-7所示。

表5-7 调用操作码

名称	参数	堆栈布局	描述
invokestatic	i1, i2	[(val1, ...)] • []	调用一个静态方法
invokevirtual	i1, i2	[obj, (val1, ...)] • []	调用一个“常规”的实例方法

（续）

名　称	参　数	堆栈布局	描　述
invokeinterface	i1, i2,count, 0	[obj, (val1, ...)] • []	调用一个接口方法
invokespecial	i1, i2	[obj, (val1, ...)] • []	调用一个“特殊”的实例方法
invokedynamic	i1, i2, 0,0	[val1, ...] • []	动态调用，见5.5节

在调用操作码中，有两个地方需要注意。第一个是invokeinterface中多出来的参数。这些参数基于历史原因和向后兼容而产生，但现在已经用不到了。在invokedynamic的参数中多出来的两个0是基于前向兼容而产生的。

另外一个是常规和特别实例方法调用之间的差别。常规调用是虚拟的。这就是说被调用的方法是在运行时按照标准的Java方法重写规则查找的。特殊调用不考虑重写。在两种情况下这很重要，即私有方法和超类方法的调用。在这两种情况下，你不想触发重写规则，所以需要不同的调用操作码处理这种情况。

5.4.8 平台操作操作码

平台操作族系的操作码包括new，用于分配新的对象实例，还有与线程相关的操作码，比如monitorenter和monitorexit。详细内容请参见表5-8。

平台操作码用来控制对象生命周期，比如创建新对象并锁住它们。一定要注意，new操作码只分配存储空间。对象构建的高层概念还包括运行构造方法内的代码。

表5-8　平台操作码

名　称	参　数	堆栈布局	描　述
new	i1, i2	[] • [obj]	为新对象分配内存，类型由指定位置的常量确定
monitorenter		[obj] • []	锁住对象
monitorexit		[obj] • []	解锁对象

在字节码这一级，构造方法被转换成带有特殊名称<init>的方法。这不能由用户代码调用，但可以由字节码调用。这便形成了一个与对象创建直接相关的不同字节码模式：new之后跟着一个dup，然后是一个调用<init>方法的invokespecial。

5.4.9 操作码的快捷形式

为了节省字节，很多字节码都有快捷形式。通常对某些局部变量的访问要比其他的访问更加频繁，所以用特殊的操作码来表示“在局部变量上直接执行常见操作”便很有价值。因此加载/存储族系中出现了aload_0和dstore_2这种操作码。

我们来检查一下其中的理论，再来看一个例子。

5.4.10 示例：字符串拼接

我们给演算类中加点料，来阐明几个稍微高级点的字节码，下面的例子会涉及字节码主要族

系中的大多数。

别忘了，Java中的字符串是不可变的。那在用+运算符把两个字符串拼在一起时发生了什么？你必须创建一个新字符串，但实际上可能不止这么简单。

看一下修改了`run()`方法之后的演算类：

```
private void run(String[] args) {
  String str = "foo";
  if (args.length > 0) str = args[0];
  System.out.println("this is my string: " + str);
}
```

这个简单方法对应的字节码为：

```
$ javap -c -p wgjd/ch04/ScratchImpl.class
Compiled from "ScratchImpl.java"

  private void run(java.lang.String[]);
    Code:
      0: ldc           #17                 // String foo
      2: astore_2
      3: aload_1
4: arraylength
5: ifle          12 #A
```

如果传入数组尺寸小于等于0，跳到指令12。

```
      8: aload_1
      9: iconst_0
     10: aaload
     11: astore_2
     12: getstatic     #19
➥ // Field java/lang/System.out:Ljava/io/PrintStream;
```

上面这行是访问`System.out`的字节码。

```
     15: new           #25                 // class java/lang/StringBuilder
     18: dup
     19: ldc           #27                 // String this is my string:
     21: invokespecial #29
➥ // Method java/lang/StringBuilder."<init>":(Ljava/lang/String;)V
     24: aload_2
     25: invokevirtual #32
➥ // Method java/lang/StringBuilder.append
➥ (Ljava/lang/String;)Ljava/lang/StringBuilder;
     28: invokevirtual #36
➥ // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
```

这些指令展示了拼接字符串的创建过程。特别是15~23表示对象创建（`new`、`dup`和`invokespecial`）的指令，但在这个例子中dup之后还有一个`ldc`（加载常量）。这种模式表明字节码调用的是一个非空构造方法，在此是`StringBuilder(String)`。

这个结果一开始可能有些出乎你的意料。你只是想把一些字符串拼在一起，但到了底层突然变成了创建额外的`StringBuilder`对象，并调用`append()`，然后又调用`toString()`。这是因为java中的字符串是不可变的。你不能通过拼接修改字符串对象，所以必须创建新的对象。`StringBuilder`是完成这个任务的便捷方法。

最后是调用相应的方法输出结果：

```
    31: invokevirtual #40
➥ // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    34: return
```

最终，输出字符串拼好了，你可以调用println()方法。因为此时栈顶部的两个元素是[System.out,<output string>]，所以这是在System.out之上调用的。就跟你在看表5-7（定义了有效的invokevirtual的堆栈布局）时所预期的一样。

要成为一名真正优秀的Java开发人员，你应该找几个自己写的类用javap运行一下，并学会识别通用的字节码模式。现在，让我们带着对字节码的简单了解，进入下一主题——Java 7中重要的新特性invokedynamic。

5.5 invokedynamic

本节主要针对Java 7中最复杂的新特性之一。尽管这个特性十分强大，但它并不是给所有开发人员准备的，它只会出现在非常高级的用例中。目前来看，这个特性是为框架开发人员和非Java语言准备的。

也就是说如果你对平台底层如何运转不感兴趣，对新的字节码细节毫不关心，请跳到小结部分或直接进入下一章，没关系的。

如果你还在，很好。接下来我们可以向你介绍invokedynamic的出现是多么不同寻常。Java 7引入了一个崭新的字节码，这在Java世界中可是从来没有过的大事件。这个字节码新秀就是invokedynamic，一种新的调用指令，是用来做方法调用的。它可以用来告诉VM必须延迟确定要调用哪个方法。也就是说VM不用像往常一样在编译或连接时就敲定所有细节。

相反，需要什么方法在运行时决定。通过调用一个辅助方法来确定应该调用哪个方法。

javac不会产生invokedynamic

在Java 7中，Java语言还不能直接支持invokedynamic，没有哪个Java表达式会被javac直接编译成invokedynamic。人们希望Java 8会增加更多的语言结构（比如默认方法）来使用这些动态能力。

invokedynamic是为非java语言准备的。添加它是为了让动态语言能够利用Java 7 VM，不过有些聪明的Java框架也找到了让invokedynamic为它们服务的办法。

我们在本节中会给出invokedynamic的工作细节，还会给出一个详细的例子——反编译一个利用新字节码的调用点。注意，要使用那些用到invokedynamic的语言和框架不一定要完全搞清楚这些内容。

5.5.1 invokedynamic如何工作

为了支持invokedynamic，Java 7又新增加了几条常量池定义。这些是在Java 6技术中无法

提供的支持。

给invokedynamic指令的索引必须指向类型为CONSTANT_InvokeDynamic的常量。这个常量上是两个16位的索引（也就是4字节）。第一个索引指向方法表（用来确定要调用什么）。它们被称为引导方法（有时简写为BSM），并且必须是静态的，还要有确定的参数签名。第二个索引指向CONSTANT_NameAndType。

从中可以看出CONSTANT_InvokeDynamic和普通的CONSTANT_MethodRef差不多，只是CONSTANT_MethodRef指明在哪个类的常量池里找寻方法，而invokedynamic调用则通过引导方法来寻找答案。

引导方法会返回一个CallSite实例，用它来接收与调用点相关的信息，并连接动态调用。调用点中有一个MethodHandle，调用点在这里起一个代理的作用，对它的所有调用实际上就是对MethodHandle的调用。[①]

invokedynamic一开始并没有目标方法（还没连接）。在第一次调用时，该点的引导方法被调用。引导方法返回一个CallSite，它被连接到invokedynamic指令上。该过程如图5-5所示。

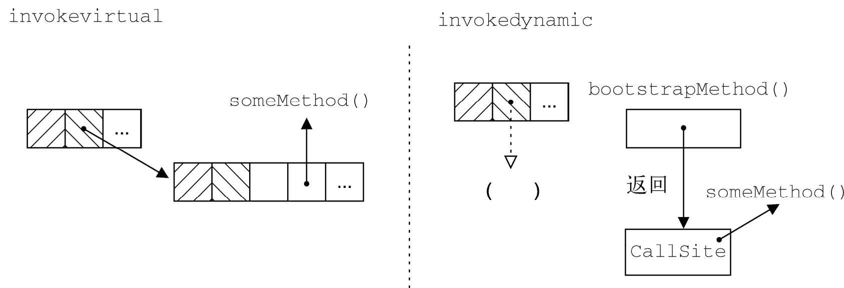


图5-5　虚拟与动态调用

连接上CallSite后，就可以调用真正的方法了，即CallSite持有的MethodHandle所指向的方法。这种设定表明JIT编译器可以像优化invokevirtual调用那样优化invokedynamic调用。下一章会讨论更多有关优化的内容。

还有一点值得注意，某些CallSite对象是可以重连的（在它们的生命期内指向不同的目标方法）。一些动态语言会大量使用这一特性。

下一节会给出一个简单的例子，我们可以看到invokedynamic调用在字节码中如何表示。

5.5.2　示例：反编译译**invokedynamic**调用

如前所述，Java 7中没有支持invokedynamic的Java语法。要得到带有动态调用指令的.class文件，你只能向字节码处理类库求助。ASM类库（http://asm.ow2.org/）就是一个不错的选择——它是一个工业级类库，在Java框架中得到了广泛应用。

① 详情请参见CallSite的Javadoc：http://cr.openjdk.java.net/~jrose/pres/indy-javadoc-mlvm/java/lang/invoke/CallSite.html。
——译者注

我们可以用这个类库构造一个包含invokedynamic指令的类，然后将其转换为字节流。这既可以写到磁盘里，也可以交给类加载器插入到运行的VM中。

一个简单的例子是让ASM产生的类包含一种invokedynamic指令的静态方法。这个方法可以由普通的Java代码调用——它封装（或隐藏）了真正调用的动态本质。作为invokedynamic开发工作的一部分，Remi Forax和ASM团队提供了一个简单的工具来产生这样的测试类。ASM是第一批完全支持新字节码的工具之一。

让我们来看一下这种封装方法的字节码：

```
public static java.math.BigDecimal invokedynamic();
  Code:
     0: invokedynamic #22,  0
➥ // InvokeDynamic #0:_:()Ljava/math/BigDecimal;
     5: areturn
```

到目前为止还没什么看头，因为复杂性主要体现在常量池中。我们来看看和动态调用相关的常量池条目：

```
BootstrapMethods:
  0: #17 invokestatic test/invdyn/DynamicIndyMakerMain.bsm:
➥ (Ljava/lang/invoke/MethodHandles$Lookup;Ljava/lang/String;
➥ Ljava/lang/invoke/MethodType;Ljava/lang/Object;)
➥ Ljava/lang/invoke/CallSite;
    Method arguments:
      #19 1234567890.1234567890
#10 = Utf8               ()Ljava/math/BigDecimal;
#18 = Utf8               1234567890.1234567890
#19 = String             #18   //  1234567890.1234567890
#20 = Utf8               _
#21 = NameAndType        #20:#10   //  _:()Ljava/math/BigDecimal;
#22 = InvokeDynamic      #0:#21    //  #0:_:()Ljava/math/BigDecimal;
```

要想完全搞清楚确实得花点心思琢磨琢磨。我们逐一来看一下。

- invokedynamic操作码在条目#22中。它指向引导方法#0和NameAndType#21。
- 在#0的BSM是类DynamicIndyMakerMain中的普通静态方法bsm()。它有BSM的正确签名。
- 条目#21给出了这个动态连接点的名称“_”，还有返回类型BigDecimal（保存在#10）。
- 条目#19是传入引导方法的静态参数。

如你所见，这里需要做很多基础工作来保证类型安全。但在运行时出错的方式仍然还有很多，但这种机制作了很大贡献，它在保留了灵活性的同时提供了安全性。

注意 BootstrapMethods方法指向方法句柄而不是直接指向方法，这提供了额外的间接性，或者说灵活性。在前面的讨论中我们并没有涉及，因为它可能会混淆正在发生的事情，对于理解这种机制如何工作并没有实质性的帮助。

到此为止，我们已经结束了对invokedynamic和字节码及类加载内部工作机制的讨论。

5.6 小结

在本章中，我们快速浏览了字节码和类加载，还解剖了类文件，并简单介绍了JVM提供的运行时环境。随着对平台内部更深入地了解，我们相信你会成为更厉害的开发者。

希望你从本章中学到如下知识：

- 类文件格式和类加载是JVM操作的核心。它们对于任何想在VM上运行的语言来说十分重要；
- 类加载的各个阶段同时保证了运行时的安全和性能特性；
- 方法句柄是Java 7主要新API之一，它是反射之外的一个可选方案；
- JVM按相关功能分为不同的族系；
- Java 7引入了`invokedynamic`：一种调用方法的新办法。

现在是时候进入下一个大主题了。通过阅读下一章，你会在性能分析方面打下坚实的基础。你将学会如何评估和优化性能以及如何充分发挥JVM核心技术的能力（比如JIT编译器，它会把字节码转换成超快的机器码）。

第 6 章

理解性能调优

6

本章内容

- ❑ 性能的重要性
- ❑ 新的垃圾收集器G1
- ❑ VisualVM：内存可视化工具
- ❑ 即时编译

糟糕的性能会“杀死”你的应用程序，使你名声扫地，在客户中的信誉大受影响。除非你具有绝对垄断地位，否则你的客户将夺门而出，直奔你的竞争对手而去。要让糟糕的性能不再糟蹋你的项目，你需要理解性能分析，还要知道如何利用它。

性能分析与调优是个非常庞大的课题，但是现在有太多处理方式都在误人子弟。所以我们准备把性能调优的秘诀透漏给你。

秘诀来了——性能调优唯一的惊天秘诀就是：*你必须量体裁衣。没有评测，就没有合适的调优。*

原因：人们总是猜不对系统变慢的是哪里。所有人都猜不对。你，我，甚至是James Gosling大神——我们总会心生偏见，并倾向于那些可能根本不存在的模式。

实际上，“我的哪些Java代码需要优化？”这个问题的答案经常是“哪个也不用，都挺好的”。

假设有一个经典（相当保守）的电子商务Web应用为注册客户提供服务。它有一个SQL数据库，一个面向Java应用服务器的Apache Web服务器，以及连接这一切的标准网络配置。系统真正的瓶颈经常是非Java部分（数据库、文件系统、网络），但不经过评测，Java开发人员永远都不会知道问题出在哪里。开发人员不去解决真正的问题，而是把时间浪费在对于改进系统性能毫无意义的代码微调上。

你希望能够回答如下几类基本问题。

- ❑ 如果你们搞了次促销，客户突然暴增十倍，系统有足够的内存来应付这种局面吗？
- ❑ 客户从应用程序中看到的平均响应时间是多长？
- ❑ 跟竞争对手比起来怎么样？

要做性能调优，你就不能猜测导致系统变慢的原因。你必须知道并且确保你真正实现性能调优的唯一办法就是性能评测。

你还需要明白性能调优不是：

- 一堆技巧和窍门；
- 秘密武器；
- 你在项目结束时撒一把的仙粉。

对“技巧和窍门”要特别小心。JVM是一个非常复杂，并经过高度优化的环境，如果脱离了上下文，这些技巧基本都没什么用，而且可能还会带来麻烦。随着JVM在代码优化方面越来越智能，它们也很快就会过时。

性能分析实际上是种试验性的科学。你可以把代码看成是某种科学试验，有输入，会产生“输出”——性能指标表明系统执行任务的效率。性能工程师的工作是研究这些输出，并找出其中的模式。所以性能调优是统计应用的一个分支，而不是一群老太婆的闲言碎语。

本章将是你的新起点，我们会向你介绍Java性能调优实战。但这是个大课题，由于篇幅有限，我们只能把你领进门，帮助你分析其中的重要原理和标志性内容。我们也会尽量解答大部分基本问题。

- 性能为什么这么重要？
- 性能分析为什么这么难？
- JVM的哪些方面会让调优变得复杂？
- 应该如何考虑和完成性能调优？
- 哪些是导致系统迟缓的最常见原因？

我们还会介绍JVM中与性能相关的两个最重要子系统：

- 垃圾收集子系统
- JIT编译器

有了这些，你就可以开始着手解决编码时遇到的实际问题了。

我们先来快速浏览一些基本词汇，以便你可以表达并框定自己的性能问题和目标。

6.1　性能术语

为了让你充分理解本章所讨论的内容，我们会给出正规的性能概念定义。下面是性能工程师词典里最重要的一些术语：

- 等待时间（Latency）
- 吞吐量（Throughput）
- 利用率（Utilization）
- 效率（Efficiency）
- 容量（Capacity）
- 扩展性（Scalability）
- 退化（Degradation）

Doug Lea讨论这些术语时都是放在多线程代码的上下文中，但我们要考虑的范围更广：从一个多线程处理器到整个集群服务器平台。

6.1.1 等待时间

等待时间是在给定工作量下处理一个任务单元所消耗的时长。通常，都是在工作量“正常”的情况下提到等待时间的。但有价值的性能评测一般都是用一张图形来显示在工作量不断增加的情况下等待时间随之改变的函数关系。

图6-1显示了在工作量增加时，某一性能指标（比如等待时间）出现了一个突发的非线性退化。这通常被称为性能肘。

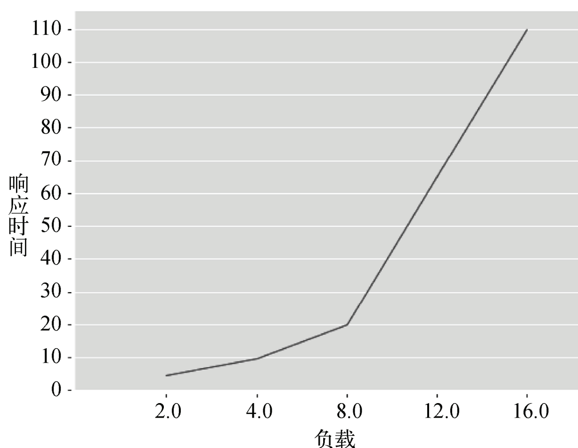


图6-1 性能肘

6.1.2 吞吐量

吞吐量是系统在限定资源、限定时长内能完成的单位工作量。用的最多的是在某一参考平台（比如指明了硬件配置、操作系统和软件环境的特定品牌服务器）上的每秒事务处理数。

6.1.3 利用率

利用率表示可用资源中用来处理工作单元(而不是清理任务或处于空闲状态)的资源百分比。人们通常会说服务器的利用率是10%，这其实是说在正常处理时间内处理工作单元的CPU百分比。注意，不同资源的利用率水平可能有非常大的差异，比如CPU和内存之间。

6.1.4 效率

系统的效率等于吞吐量除以所用资源。一个用更多资源产生相同吞吐量的系统效率更差。

比如比较两个集群方案。如果为了达到相同的吞吐量，方案A需要的服务器数量是方案B的两倍，则方案A的效率是B的一半。

别忘了，资源也可以用成本来衡量——如果方案X的成本是方案Y的两倍或需要两倍的员工运行生产环境，则方案X的效率是Y的一半。

6.1.5　容量

容量是任一时刻能通过系统的工作单元（比如事务）数量。也就是在特定的等待时间或吞吐量下，能够得到同步处理的工作单元数量。

6.1.6　扩展性

当系统得到更多资源时，它的吞吐量或等待时间会发生变化。这种发生在吞吐量或等待时间上的变化就是系统的扩展性。

如果方案A可用的服务器数量翻倍，它的吞吐量也能翻倍，那我们就说它实现了完美的线性扩展。在大多数情况下，完美的线性扩展很难达到。

还应该注意，系统的扩展性取决于很多因素，而且这种扩展性还是变化的。系统能以线性方式向上扩展到某一点，然后开始退化。这是另外一种性能肘。

6.1.7　退化

如果在不增加资源的情况下增加工作单元或网络系统的客户端，一般等待时间或吞吐量都会发生变化。这是系统在负载增加时出现的退化。

正面退化与负面退化

在正常情况下，退化是负面的。也就是说给系统增加工作单元会对性能产生负面影响，比如导致处理等待时间变长。但某些情况下退化也有可能是正面的。

比如说，如果过重的负载导致系统某些部分超过了阈值，迫使系统切换到高性能模式，这会让系统工作效率更高，缩短处理时间，尽管还要完成更多工作。JVM是个动态性非常强的运行时系统，并且有几部分可以达成这种效果。

前面这些术语是最常用的性能指标，当然还有其他一些重要的指标，但这些是指导系统性能调优的基本统计数据。在下一节中，我们会给出一个以密切关注这些数值为基础，并尽可能定量的性能调优方法。

6.2　务实的性能分析法

许多开发人员在接到性能分析任务时，脑子里都不清楚他们要通过分析得到什么。所有开发人员或经理在开始做这件事时经常只是模模糊糊地感觉代码“应该跑得更快”。

但这是彻底的倒退。要进行真正有效的性能调优，在开始做任何技术类工作之前，你应该先认真考虑下面这些问题并找出答案。

- 你正在测量的代码有哪些可观测的环节？
- 如何测量那些可观测环节？
- 这些可观测环节的目标是什么？
- 你怎么判断性能调优是否做好了？
- 性能调优可接受的最大支出是多少（按开发人员投入的时间和增加的代码复杂度计算）？
- 在优化的过程中，哪些东西是你不能舍弃的？

最重要的，也是我们要反复强调的，就是你必须测量。你至少得测量一个可观测环节，才算得上是在做性能分析。

当你开始测量代码，便经常会发现事情并非你想的那样。很多性能问题的根源可能是一个丢失的数据库索引，或者有争议的文件系统锁。在优化代码时，你应该时刻牢记代码很可能不是问题的关键。为了定量分析问题，你首先需要知道自己在测量什么。

6.2.1 知道你在测量什么

做性能调优必须测量一些东西。如果你没有测量可观测环节，就不能算做性能调优。坐在那里盯着代码，希望脑子里蹦出一个可以更快解决问题的方法，这可不是性能分析。

提示 要成为优秀的性能工程师，你必须知道平均数、中位数、模式、方差、百分位数、标准差、样本大小、正态分布等这样一些术语。如果还不熟悉这些概念，最好现在就到网上搜搜，如果有必要的话，认真看看搜出来的内容。

做性能分析最重要的是知道哪个可观测环节（上节介绍的）最重要。你应该总是把测量结果、目标和结论跟一个或多个基本可观测环节结合起来。

这里有些常见的可观测项，都是性能调优的好对象。

- 方法`handleRequest()`运行所需的平均时间（启动完成之后）。
- 并发客户端数量为10时，系统等待时间的第90个百分位数。
- 把并发用户数从1增长到1000时，响应时间的退化。

以上这些都是工程师想要测量的代表性数值，并很有可能需要优化。想得到准确又有用的数值，必须掌握基本的统计学知识。

知道你要测量什么，对数值的准确性有信心是性能调优的第一步。但模糊或随意的目标通常没什么好结果，性能调优也是如此。

6.2.2 知道怎么测量

要精确确定一个方法或其他代码片段运行需要多长时间，只有两种方法：

- 直接测量，在类源码中插入测量代码；
- 在类加载时把类转换成受测类。

大多数简单直接的性能测量技术都依赖于以上其中一种或全部技术。

还应该提一下JVM工具接口（JVMTI），用它可以创建非常复杂的分析器，但它也有缺陷。它需要性能工程师编写本地代码，并且它产生的分析数值本质上是统计平均值，而不是直接测量结果。

直接测量

直接测量是最容易理解的技术，但它是侵入式的。最简单的是像下面这种形式：

```
long t0 = System.currentTimeMillis();
methodToBeMeasured();
long t1 = System.currentTimeMillis();
long elapsed = t1 - t0;
System.out.println("methodToBeMeasured took "+ elapsed +" millis");
```

这段代码会输出`methodToBeMeasured()`精确到毫秒的运行时长。很不方便的是，你要到处添加这种代码，而且随着测量结果不断增多，代码很容易被数据淹没。

除此之外还有其他问题，如果`methodToBeMeasured()`运行时长不足一毫秒会出现什么情况？稍后我们就会看到，此外还值得注意的是冷启动效果——后运行的方法可能比先运行的快。

通过类加载自动测量

我们在第1章和第5章讨论过如何把类编译成可执行程序。其中一个关键步骤是在加载字节码时进行转换。这个特性非常强大，是很多现代Java平台的核心技术。其中一个简单的例子就是方法的自动测量。

在这种方法中，特殊的类加载器加载`methodToBeMeasured()`所属类，在方法开始和结束的地方加上记录方法进入和退出时间的字节码。这些时间通常会被写入共享的数据结构，由其他线程访问。这些线程一般会将数据写入日志文件，或者通过网络交给负责处理原始数据的服务器。

很多高端的性能监测工具（比如OpTier CoreFirst）都是以这项技术为核心的。但在编写本书时，这个市场上似乎还没有开源工具。

注意　我们会在后面讨论到，Java 方法开始时需要进行解释，然后才切换到编译模式。要得到真正的性能指标结果，你必须去掉解释模式占用的时间，因为它们会严重扭曲真实结果。后面还会给出更多细节，告诉你如何确定方法切换为编译模式的时间。

你可以用这两项技术（其一或全部）找出某一方法执行所需的时长。下一个问题，完成调优之后，你想得到什么样的数值？

6.2.3　知道性能目标是什么

清晰的目标能让人注意力集中，所以了解和传达优化的最终目标（知道要测量什么）至关重要。大多数情况下，这个目标简单而明确，比如：

- 将10个并发用户的端到端等待时间的第90个百分位数减少20%；
- 将`handleRequest()`的平均等待时间减少40%，方差减少25%。

在一些更复杂的情况中，目标可能由几个相关的性能目标共同构成。你要知道，你所测量和想要优化的独立可观测项越多，调优工作就会变得越复杂。优化一个性能目标可能会对其他性能目标产生负面影响。

有时，在设定目标之前你很有必要做些初步分析，比如在确定要让方法运行得更快这一目标之前，应该先确定哪些方法最重要。这很好，但经过初步探索后，你最好停下来再确认一下目标，然后再达成它们。开发人员非常爱犯只顾低头拉车，不顾抬头看路的错误。

6.2.4　知道什么时候停止优化

理论上来说，知道什么时候停止优化并不难——达成目标之时就是任务完成之日。然而实际中人们很容易陷入性能调优的泥淖。如果事情进展顺利，你肯定想要继续前进并做得更好。而如果不太顺利，你为了达成目标就会不断尝试新策略。

要想知道什么时候停止优化，你需要对目标有清醒的认识并理解它们的价值。能达成性能目标的90%通常就足够了，你还可以利用节省下来的时间去做些别的事。

还要考虑一点，你要看看有多少工作投入到了极少用到的代码路径上。通过优化代码来减少程序运行时长的1%（甚至更少）完全是在浪费时间，但奇怪的是做这种事儿的开发人员数量惊人。

至于该优化什么，这里有一组非常简单的指导规则。你可能需要根据自身情况进行调整，但它们的适用范围很广泛：

- 优化那些重要，而不是最容易的代码。
- 首先优化那些最重要（通常是调用最频繁）的方法。
- 在遇到那些唾手可得的优化时，把它办了，但要清楚代码的调用频率。

最后再做一轮测量工作。如果还没达成性能目标，你就需要清查一下，看看离命中目标还有多大差距，以及取得的成绩是不是已经对整体性能产生了你所期望的影响。

6.2.5　知道高性能的成本

所有性能调整都贴着价签。

- 分析和优化代码要占用的时间（在任何软件项目中，开发人员的时间基本都是最大的开支）。
- 所做的调整可能会引入额外的技术复杂度（也有简化代码的性能优化，但它们不是主流）。
- 为了让主处理线程运行得更快，可能会引入额外的线程来执行辅助任务，但这些线程可能会在负载较高时对系统整体产生不可预料的影响。

不管是什么价签，你都要重视，并尽量在完成第一轮优化之前找到它们。

这有助于你了解提高性能的最大可接受成本。这个成本可能是设定开发人员调优的时间限制，额外的类数或代码行数。比如说，开发人员决定花在优化上的时间不能超过一个星期，或者因优化而生的类增长不应该超过100%（即大小变成原来的两倍）。

6.2.6 知道过早优化的危险

关于优化，Donald Knuth有段著名的评论：

> 程序员浪费了大量时间考虑，或担心程序中无关紧要部分的速度，并且那些尝试改进效率的行为实际上有很强的负面影响……过早优化是万恶之源。①

这段话在业内引起了广泛争论，而且人们通常只记住了最后一句。这之所以令人感到遗憾，有如下原因。

- 在评论的前段，Knuth含蓄地提醒我们要测量，没有测量就不能确定程序的关键部分。
- 我们再次提醒你，可能不是代码导致等待时间过长——环境中的其他部分也会产生等待时间。
- 在完整的评论中，很容易看出Knuth是在谈论那些有意识的、齐心协力的优化。
- 这段评论的简短版让它变成了不良设计或糟糕执行选择的相当巧合的借口。

有些优化体现在良好的编码风格上：

- 不要分配不需要的对象。
- 如果再也不需要调试日志，就去掉它。

我们在下面的代码中加了一个检查，看日志对象是否处理调试日志。这种检查被称为日志守卫。如果日志子系统被设置为不处理调试日志，这段代码就不会构造日志消息，省掉了为了日志消息而调用`currentTimeMillis()`和构造`StringBuilder`对象的开销。

```
if (log.isDebugEnabled()) log.debug("Useless log at: "+
System.currentTimeMillis());
```

但如果调试日志真的没有用，我们可以把这段代码一并去掉，就能再节省两个处理器周期（日志守卫的开销）。

性能调优的工作之一就是从一开始就写出质地优良、高效运行的代码。更好地认识Java平台，知道它的底层运行机制（比如理解在合并两个字符串时隐含的对象分配），并在编码时考虑到性能问题，才能写出更好的代码。

现在我们有了框定性能问题和目标的基本词汇，还有如何解决问题的方法大纲。但我们还没解释为什么这是软件工程师会遇到的问题，以及这种需求来自哪里。要弄懂这个，我们有必要简单了解一下硬件的世界。

6.3 哪里出错了？我们担心的原因

在几年前，性能问题看起来并不重要。时钟速度不断上升，所有软件工程师只要多等几个月，哪怕是写得很烂的代码，也能借助节节攀升的CPU速度表现出优异性能。

① Donald E. Knuth，“带go to语句的结构化编程”，计算调查，6，no.4（1974年12月）。http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf。

那么，事情怎么会错得这么离谱？为什么时钟速度的提升不再那么快了？更让人担忧的是，为什么有3GHz芯片的电脑看起来比2GHz芯片的快不了多少？行业中软件工程师需要考虑性能问题的这种趋势是从哪里来的？

我们会在本节讨论引导这股趋势的力量，以及连最纯粹的软件开发人员也需要了解一点硬件知识的原因。我们会为本章剩下的内容打好基础，让你理解JIT编译的概念和一些有深度的例子。

你可能听说过“摩尔定律”。很多开发人员都知道这个定律与提高计算机速度有关，但对于具体细节不甚了了。我们来解释一下它到底是什么意思，以及它在不久的将来可能带来的影响。

6.3.1　摩尔定律——过去和未来的性能趋势

摩尔定律是Gordon Moore提出来的，他是Intel的创始人之一。该定律最常见的形式之一是：晶片上的晶体管数量每两年翻一番是合算的。

这个定律实际上是对计算机处理器（CPU）发展趋势的一种看法，基于Gordon Moore在1965年发表的一篇论文，最初是对未来10年进行预测——也就是直到1975年。而这一预测直到现在仍然能够应验（据预测其在2015年以前都有效），实在值得称道。

我们在图6-2中绘制了Intel x86家族从1980年发展到2010年的i7整个历程中的一些真实CPU。图中显示了不同时期发布的芯片所集成的晶体管数量。

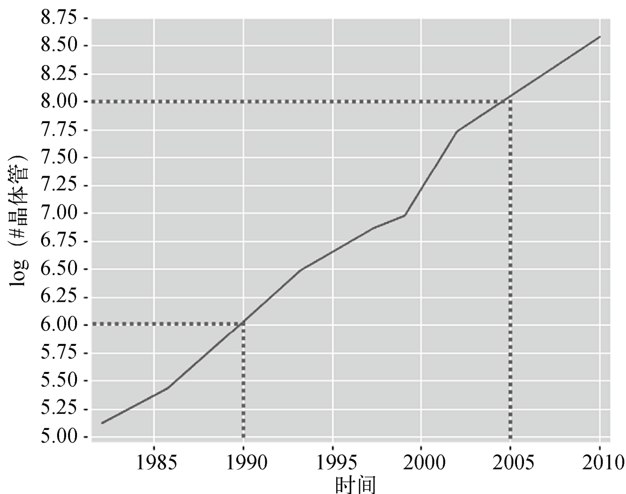


图6-2　随时间变化的晶体管数量对数线性图

这是一个对数线性图，所以y轴上的每次增长都是前一个值的10倍。如你所见，这条线基本是直的，而且每隔六或七年就会穿过一个垂直层级。这证明了摩尔定律的准确性，因为六或七年增长十倍和每隔两年翻一番基本是一致的。

图中y轴的刻度是对数，这就是说Intel在2005年生产的主流新品大概有一亿个晶体管。这是1990年生产的芯片上的晶体管数量的100倍。

一定要注意摩尔定律特别谈到了晶体管数量。要知道，单凭摩尔定律不足以让软件工程师继续从硬件工程师那里得到好处，理解这一点是最基本的要求。

注意　晶体管数量和时钟速度不是一回事儿，人们一般会认为更高的时钟速度就意味着更好的性能，其实这是一种过于草率的想法。

摩尔定律在过去很有指导意义，并且在未来一段时间内应该仍然准确（有不同的估算，但在2015年前看起来是合理的）。但摩尔定律计算的是晶体管数量，用这个数值来指导开发人员提高代码性能越来越不靠谱。实际上，你会发现情况要更复杂。

在实际操作中，性能是由一系列因素共同决定的，而且每个因素都很重要。然而如果硬让我们从里面挑一个，应该是定位指令与数据运行速度的相关性。这个概念对性能非常重要，我们会深入了解。

6.3.2　理解内存延迟层级

计算机处理器要处理数据。如果它要处理的数据到不了，CPU时钟多快都没用——它只能等着，执行空操作（NOP），在数据到来之前基本就处于停转状态。

也就是说在解决延迟时，最重要的两个问题是，“CPU核心要处理的数据的最近的复本在哪里？”，还有“把它送到核心能用的地方需要多长时间？”主要有以下几种答案。

- 寄存器：这是CPU上的内存地址，随时可用。这部分内存是指令直接操作的。
- 主存：这一般是DRAM。访问时间在50纳秒左右（关于如何使用处理器缓存避免这段延迟请参见后续内容）。
- 固态磁盘（SSD）：访问这种磁盘所需的时间不足0.1毫秒，但跟传统硬盘比起来，它们要便宜一些。
- 硬盘：访问这种磁盘并把数据加载到主存中大概需要5毫秒。

摩尔定律预测了晶体管数量的指数级增长，这对内存也有好处——内存访问速度也能以指数级增长。但这两种指数并不相同。内存速度的提升比CPU晶体管数量的增长要慢得多，这意味着核心迟早会因为没有相关数据可以处理而落入空闲状态。

为了解决这个问题，在寄存器和主存之间引入了缓存。缓存是少量更快的内存（SRAM，而不是DRAM）。这种更快的内存成本要比DRAM高很多（无论是金钱还是晶体管），所以计算机不全用SRAM作为内存。

缓存分为一级缓存（L1）和二级缓存（L2）（某些机器还有L3），数值表明缓存到CPU的距离（越近越快）。我们会在6.6节（在JIT编译上）详细讨论缓存，并给出一个例子来表明L1缓存对运行代码的重要影响。图6-3展示了L1和L2缓存比主存快多少。之后，我们还会给出一个例子来阐明这些速度差异对运行代码的性能有什么影响。



图6-3 寄存器、处理器缓存和主存的相对访问时间

除了增加缓存，20世纪90年代到21世纪早期大量使用了另外一种技术解决内存延迟的问题，就是增加处理器的功能，这使得处理器越来越复杂。即便CPU处理能力和内存延迟之间的差距越来越大，也仍然采用复杂的硬件技术来保证CPU有数据可以处理，比如指令级并行（ILP）和芯片多线程（CMT）。

这些技术的出现消耗了CPU晶体管预算中的大部分，并且它们使真实性能收益递减。这一趋势导致了新观点的出现，即在未来设计带有多个（或很多）核心的CPU芯片。

这意味着未来的性能和并发密切相关——主要办法之一就是通过拥有更多核心让系统整体性能得到提升。那样，即便有一个核心在等待数据，其他核心也可以继续工作。这种关系十分重要，所以我们要一再提起。

- 将来的CPU是多核的。
- 性能和并发绑在一起变成了相同的关注点。

Java程序员除了要关注硬件，还要注意JVM特性带来了额外的复杂性。下一节我们就来看一下这些内容。

6.3.3 为什么Java性能调优存在困难

在JVM或其他任何受控运行时环境上做性能调优天生就比在非受控环境下做调优困难。这是因为C/C++程序员几乎所有事情都要自己做。OS只提供很少的服务，比如基本的线程调度。

在受控系统中，基本观点是让运行时来控制环境，不用开发人员自己处理所有细节。这能提高程序员的生产率，但要放弃某些控制权。另外一种选择是放弃受控运行时提供的所有便利，但和性能调优所做的工作相比，这个代价实在太高了。

造成调优困难的平台特性主要是：

- 线程调度；
- 垃圾收集（GC）；
- 即时（JIT）编译。

这些特性能以很巧妙的方式交互。例如，编译子系统用计时器来决定编译哪个方法。也就是说等待编译的候选方法集可能会受到调度和GC等特性的影响。每次运行时所编译的方法可能都不同。

正如你在本节中看到的，准确测量是性能分析决策过程的关键。如果你决定认真对待性能调优，那么理解Java平台中处理时间的细节和限制就非常有用。

6.4 一个来自于硬件的时间问题

你有没有想过计算机里的时间存在哪里以及在哪里处理？我们都知道硬件最终负责跟踪时间，但事实可能不像你想的那么简单。

为了进行性能调优，你需要对时间如何工作有深刻的认识。为此我们先从底层硬件开始讨论，然后探讨Java如何与这些子系统集成，最后介绍`nanoTime()`方法的复杂性。

6.4.1 硬件时钟

在基于x64的机器里有四种不同的硬件时间源：RTC、8254、TSC以及HPET。

实时时钟（RTC）基本上和便宜的电子表（基于石英晶体）里找到的电子器件一样，在系统断电时由主板上的电池供电。系统在启动时就是从它那里得到时间的，不过很多机器在OS启动过程中会通过网络时间协议（Network Time Protocol，NTP）跟网络上的时间服务器同步。

所有古董都曾是新东西

实时时钟这个名字现在看来十分不恰当——在20世纪80年代它刚出现时确实被认为是实时的，但现在它的准确度对于关键应用来说已经不够用了。以“新”或“快”命名的创新经常是这种结局，比如巴黎的Pont Neuf（“新桥”）。它建于1607年，现在已经是巴黎市内最古老的桥了。

8254是可编程计时芯片，也是始祖级的东西。它的时钟源是一个119.318kHz的晶体，这个频率是NTSC彩色副载波频率的三分之一，这也是它返回到CGA图形系统的原因。它曾经为OS调度器提供定期时点（用于时间片），但现在已经有其他时间源（或者不再需要）了。

下面介绍应用最广泛的现代计时器——时间戳计时器（TSC）。基本上，这是一个跟踪CPU运行了多少个周期的CPU计数器。乍看起来它似乎很适合做时钟。但这个计数器是跟CPU的，并且在运行时可能会受到节能或其他因素的影响。也就是说，不同的CPU会互相偏离，也不能跟钟表时间保持一致。

最后还有高精度事件计时器（HPET）。这种计时器是最近几年才出现的，有助于人们用较老的时钟硬件更好地计时。HPET使用至少10MHz的计时器，所以其精度至少应该是1μs——但它并不是在所有硬件上都可用，也不是所有操作系统都支持。

如果这些内容看起来有点乱，那是因为它们本来就乱。好在Java平台提供了可以使用它们的工具——它把对硬件和OS支持的依赖隐藏到特定的机器配置里。然而试图隐藏依赖项的做法并没有完全成功。

6.4.2 麻烦的`nanoTime()`

Java中有两个获取时间的方法：`System.currentTimeMillis()`和`System.nanoTime()`，后面一个用于测量比毫秒更精确的时间。表6-1总结了它们两个的主要差异。

表6-1 Java内置时间获取方法的比较

currentTimeMillis()	**nanoTime()**
解析度为毫秒级	纳秒级引用
几乎所有情况下都跟钟表时间相符	可能偏离钟表时间

如果表6-1中对nanoTime()的描述让它看起来有点像计时器，那就对了，因为如今在大多数操作系统上，它的时间源都是CPU计数钟——TSC。

nanoTime()的输出是相对于某个固定时间的。也就是说必须用它记录间隔期，用nanoTime()的返回结果减去之前调用得到的返回结果。下面这段代码来自后面的一个研究案例，恰好表明了这种情况：

```
long t0 = System.nanoTime();
doLoop1();
long t1 = System.nanoTime();
...
long el = t1 - t0;
```

el是doLoop1()执行所用的时间(以纳秒为单位)。

要在性能调优中正确使用这些方法，必须对nanoTime()的行为有所了解。代码清单6-1输出了毫秒计时器和纳秒计时器（通常由TSC提供）之间的最大偏离。

代码清单6-1 时间偏离

```
private static void runWithSpin(String[] args) {
  long nowNanos = 0, startNanos = 0;
  long startMillis = System.currentTimeMillis();
  long nowMillis = startMillis;

  while (startMillis == nowMillis) {        <-- 将startNanos在毫秒边界上对齐
    startNanos = System.nanoTime();
    nowMillis = System.currentTimeMillis();
  }

  startMillis = nowMillis;
  double maxDrift = 0;
  long lastMillis;

  while (true) {
    lastMillis = nowMillis;
    while (nowMillis - lastMillis < 1000) {
      nowNanos = System.nanoTime();
      nowMillis = System.currentTimeMillis();
    }

    long durationMillis = nowMillis - startMillis;
    double driftNanos = 1000000 *
(((double)(nowNanos - startNanos)) / 1000000 - durationMillis);
    if (Math.abs(driftNanos) > maxDrift) {
      System.out.println("Now - Start = "+ durationMillis
+" driftNanos = "+ driftNanos);
      maxDrift = Math.abs(driftNanos);
    }
  }
}
```

这段代码会输出可观测到的最大偏离，并且证明其表现与操作系统的相关度很高。下面是Linux上的一段输出：

```
Now - Start = 1000 driftNanos = 14.99999996212864
Now - Start = 3000 driftNanos = -86.99999989403295
Now - Start = 8000 driftNanos = -89.00000011635711
Now - Start = 50000 driftNanos = -92.00000204145908
Now - Start = 67000 driftNanos = -96.0000033956021
Now - Start = 113000 driftNanos = -98.00000407267362
Now - Start = 136000 driftNanos = -98.99999713525176
Now - Start = 150000 driftNanos = -101.0000123642385
Now - Start = 497000 driftNanos = -2035.000012256205
Now - Start = 1006000 driftNanos = 20149.99999664724
Now - Start = 1219000 driftNanos = 44614.00001309812
```

注意**driftNanos**从**-2035**到**20149**出现了一个非常大的跳跃

这里还有一个装在相同硬件上的老Solaris上的输出结果：

```
Now - Start = 1000 driftNanos = 65961.0000000157
Now - Start = 2000 driftNanos = 130928.0000000399
Now - Start = 3000 driftNanos = 197020.9999999497
Now - Start = 4000 driftNanos = 261826.99999981196
Now - Start = 5000 driftNanos = 328105.9999999343
Now - Start = 6000 driftNanos = 393130.99999981205
Now - Start = 7000 driftNanos = 458913.9999998224
Now - Start = 8000 driftNanos = 524811.9999996561
Now - Start = 9000 driftNanos = 590093.9999992261
Now - Start = 10000 driftNanos = 656146.9999996916
Now - Start = 11000 driftNanos = 721020.0000008626
Now - Start = 12000 driftNanos = 786994.0000000497
```

间隔很平滑

注意看最大值的增长，在Solaris上很稳定，而在Linux上相当一段时间内看起来都OK，然后出现了大的跳跃。我们在选择示例代码时相当认真，尽量避免创建额外的线程，甚至对象，以将平台的干预降到最低（比如说，没有对象的创建就意味着不会做垃圾收集），但即便如此，我们还是能看到JVM的影响。

最终证实Linux时序上出现的跳跃是由不同CPU上的TSC计数器之间的差异造成的。JVM会定期挂起正在运行的Java线程，并将它迁移到不同核心上。所以程序代码会见到不同CPU计数器上的差异。

这就是说对于间隔较长的时间，`nanoTime()`基本上是不可信的。只能用它测量较短的时间间隔，较长（宏观）的时间间隔应该用`currentTimeMillis()`重新校准。

要充分掌握性能调优，即要有扎实的测量理论，还需要知道实现细节。

6.4.3　时间在性能调优中的作用

要做好性能调优，你必须知道该如何解读代码运行期间得到的测量记录，也就是说你必须明白在Java平台上得到的时间测量结果的局限性。

精确度

时间的量通常被冠以与其最接近的某一刻度单位。这被称为测量的*精确度*。比如说，测量时间经常精确到毫秒。如果重复围绕同一数值进行测量，其结果范围很小，则该计时器是精确的。

精确度是对给定测量中所包含的随机噪音量的度量。我们假定对一段代码的测量结果是正态分布的。那么精确度通常是宽度为95%的置信区间。

准确度

测量（指测量时间）的*准确度*是取得接近真实值的测量结果的能力。实际上，真实值一般不可知，所以准确度可能比精确度更难确定。

准确度是对测量中的系统性错误的度量。可能存在准确但不太精确的测量结果（所以基本读数是正确的，但有随机的环境噪音）。也可能存在精确但不准确的结果。

理解测量结果

一个精确到纳秒的时间间隔测量结果是用准确度为1微秒的计时器测量的，其值为5945纳秒，那真实值应该介于3945～7945纳秒之间（95%的可能性）。当心那些看起来过于精确的数值，必须随时对测量结果的精确度和准确度进行检查。

粒度

系统真正的*粒度*是最快计时器的频率——很可能是10纳秒范围内的中断计时器。这有时被称为*可辨别能力*，可以肯定“几乎一起发生，但时间不同”是两个事件最短的发生间隔。

在我们跨过操作系统、虚拟机和类库代码的不同层面时，已经不太可能辨别这些极短的时间间隔。在大多数情况下，应用程序开发人员是得不到这些特别短的时间间隔的。

分布式网络计时

我们对性能调优的大部分讨论都是以单机上的系统为中心的。但你应该知道，当涉及网络上的系统调优时，会有一些特别的问题。网络上的同步和计时并不容易，而且不仅仅是在互联网上，即便是以太网也会出现这些问题。

详细讲解分布式网络计时超出了本书的范围，但你应该知道，通常来说，很难得到用于跨越几台机器的工作流的准确时序。另外，即便NTP这样的标准协议对于高精度工作来说准确度也不够。

在开始讨论垃圾收集之前，我们先看一个前面提到过的例子——缓存对代码性能的影响。

6.4.4 案例研究：理解缓存未命中

对于很多吞吐量较高的代码来说，影响性能的一个主要因素就是一级缓存未命中的数量。

代码清单6-2中的代码操作1MB的数组，并输出执行两个循环中之一所用的时间。在第一个循环中，每隔16个条目对`int`数组中的元素加1。一级缓存的一个缓存行中通常有64个字节（在32位JVM上，Java的int是4个字节），所以这意味着每次会读取一个缓存行（64=16*4）。

代码清单6-2 理解缓存未命中

```
public class CacheTester {
  private final int ARR_SIZE = 1 * 1024 * 1024;
  private final int[] arr = new int[ARR_SIZE];

  private void doLoop2() {
    for (int i=0; i<arr.length; i++) arr[i]++;
  }
```



处理每个条目

```
  private void doLoop1() {
    for (int i=0; i<arr.length; i += 16) arr[i]++;        ◁ 处理每个缓存行
  }

  private void run() {
    for (int i=0; i<10000; i++) {                          ◁ 代码热身
      doLoop1();
      doLoop2();
    }
    for (int i=0; i<100; i++) {
      long t0 = System.nanoTime();
      doLoop1();
      long t1 = System.nanoTime();
      doLoop2();
      long t2 = System.nanoTime();
      long el = t1 - t0;
      long el2 = t2 - t1;
      System.out.println("Loop1: "+ el +" nanos ; Loop2: "+ el2);
    }
  }

  public static void main(String[] args) {
    CacheTester ct = new CacheTester();
    ct.run();
  }
}
```

注意，在你得到准确结果之前应该让代码热热身，以便让JVM对你感兴趣的方法进行编译。我们会在6.6节讨论更多与代码热身相关的内容。

第二个循环，`doLoop2()`给数组中的每个元素加1，所以看起来它做的工作是`doLoop1()`的16倍。下面是在笔记本上运行这段代码得到的结果：

```
Loop1: 634000 nanos ; Loop2: 868000
Loop1: 801000 nanos ; Loop2: 952000
Loop1: 676000 nanos ; Loop2: 930000
Loop1: 762000 nanos ; Loop2: 869000
Loop1: 706000 nanos ; Loop2: 798000
```

计时子系统的疑难杂症

结果中的所有纳秒值都很整齐，全是一千的整数倍。这表明底层系统调用（`System.nanoTime()`最终所调用的）仅仅返回了一个微秒整数值——一微秒是1000纳秒。因为这个结果是在Mac笔记本上得到的，所以我们猜测在OS X的底层系统调用只有微秒级的精度，实际上，它调用的是`gettimeofday()`。

从这个结果来看，`doLoop2()`所用的时长不是`doLoop1()`的16倍。这表明内存访问在总体性能配置中占有支配性地位。`doLoop1()`和`doLoop2()`读取缓存行的次数相同，而修改数据所用的CPU周期只占整体时间的一小部分。

我们先来回顾下Java时间系统的要点。

- 大多数系统内部都有几个不同的时钟。
- 毫秒计时器是安全可靠的。
- 更高精度的时间需要仔细处理以防止出现偏离。
- 你需要知道计时测量的精确度和准确度。

我们下一个将要讨论的是Java平台的垃圾收集子系统。这是性能的决定性因素中非常重要的一部分，并且它有很多可调节的部分，对于做性能分析的开发人员来说都可以成为非常重要的工具。

6.5　垃圾收集

内存自动管理是Java平台最重要的组成部分之一。在出现Java和.NET这样的托管平台之前，开发人员把大部分时间都用在追踪不完善的内存处理引发的bug上了。

然而近年来，内存自动分配技术发展的如此先进可靠，已经变得让人无法察觉了，因此大部分Java开发人员不知道Java平台的内存管理是如何完成的，不知道可以使用哪些选项，也不知道如何在框架限定内进行优化。

这说明Java的做法取得了成功。大多数开发者不知道内存和GC系统的细节是因为他们没必要知道。虚拟机在这方面做得非常棒，在处理大多数应用时都不用特别调整，所有大多数应用从没调整过。

本节我们将讨论在确实需要做些调整的情况下你能做什么。我们会给出基本原理，解释为了运行Java进程该如何处理内存，并探索标记和清除集合的基础，再讨论两个工具——`jmap`和`VisualVM`。最后介绍两个收集器——并发标记清除（Concurrent Mark-Sweep，简称CMS）和新的垃圾优先（Garbage First，简称G1）收集器。

也许你有个服务器端程序耗光了内存，或者承受着长时间中断的痛苦。在6.5.3节讨论`jmap`时，我们将会告诉你一个查看类是否占用大量内存的简单办法。我们还会教你使用控制虚拟机内存配置的选项开关。

先从基本算法开始吧。

6.5.1　基本算法

标准的Java进程既有栈又有堆。栈保存原始型局部变量（引用型局部变量会指向以堆方式分配的内存）。堆保存要创建的对象。图6-4展示了各种类型变量存储的位置。

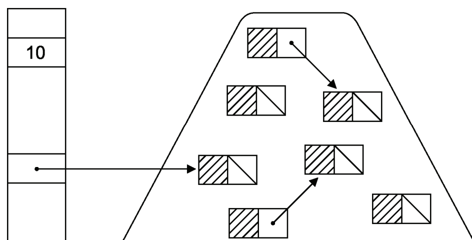


图6-4　堆和栈中的变量

注意，对象的原始型域仍然分配在堆内的地址上。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，应用程序中代码已经不再使用它了。

6.5.2　标记和清除

标记和清除是最简单、也是出现最早的垃圾收集算法。业内还有其他内存自动管理技术，比如Perl和PHP等语言采用的引用计数[①]，有人说它更简单，但它是不需做垃圾收集的方案。

最简单的标记和清除算法会暂停所有正在运行的线程，并从一组“活”对象——在任何用户线程的任何堆栈帧中存在引用（不管是局部变量、方法参数、临时变量，还是某些非常少见的情况）的对象——开始遍历其引用树，标记出遍历路径上的所有活对象。遍历完成后，所有没被标记的都被当做垃圾，可以回收（清除）。注意，被清除的内存不会还给操作系统，而是还给JVM。

Java平台对基本的标记清除方法进行了改进，采用“分代式垃圾收集”。在这种方法中，会根据Java对象的生命周期将堆内存划分为不同的区域。在对象的生存期内，对它的引用可能指向内存中几个不同区域（如图6-5所示）。在垃圾收集过程中，可能会将对象移动到不同区域。

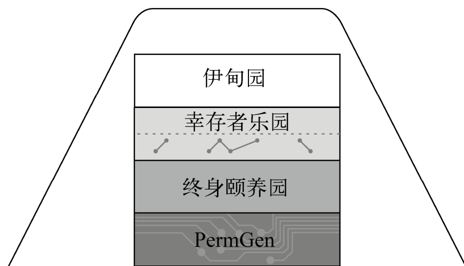


图6-5　内存中的伊甸园、幸存者乐园、终身颐养园和PermGen区

这样做是因为根据对系统运行时期的研究，发现对象的生存期或者较短，或者很长。Java平台把堆内存划分为不同区域可以充分利用对象生命周期的这种特点。

出现时长不确定的暂停怎么办？

Java和.NET经常受到这样的批评：标记和清除式的垃圾收集不可避免地会导致世界停转（所有用户线程都必须停止），而且这种暂停的时长是不确定的。

其实这个问题被夸大了。对于服务器端软件来说，应用程序不会在意垃圾收集引起的暂停。为了避免暂停或完全收集而精心制作解决方案完全是凭空想象——除非经过认真分析，发现全内存收集时间真的存在问题，才应该避免。

① 引用计数就是为每个内存对象维护一个引用数值，当有新的引用指向该对象时则将其引用计数加一，销毁时则减一。当引用计数为零时就收回该对象占用的内存资源。这种方式虽然简单，但存在两个问题：每次内存对象被引用或引用被销毁时必须修改引用计数，造成整体性能消耗；出现循环引用时难以处理。——译者注

1. 内存区域

JVM为存储不同生命周期阶段的对象将内存分成了几个不同区域。

- 伊甸园——伊甸园是对象最初降生的堆区域，并且对大多数对象来说，这里是它们唯一存在过的区域。
- 幸存者乐园——这里通常有两个空间（或者也可以认为是被分成两半的一个空间）。从伊甸园幸存下来的对象会被挪到这里。它们有时候被称为从何而来和到哪里去。除非正在执行垃圾收集，否则总有一个幸存者空间是空的，原因会在后面给出。
- 终身颐养园——终身制空间（即老一代）是那些“足够老”的幸存对象的归宿（从幸存者空间挪过来的）。在年轻代收集过程中是不会碰终身制内存的。
- PermGen——这是为内部结构分配的内存，比如类定义。PermGen不是严格的堆内存，并且普通的对象最后不会在这里结束。

就像前面提到的，这些内存区域的垃圾收集方式也不尽相同。具体来说有两种方式：年轻代收集和完全收集。

2. 年轻代收集

年轻代收集只会清理“年轻的”空间（伊甸园和幸存者乐园）。其过程相当简单。

- 在标记阶段发现的所有仍然存活的年轻对象都会被挪走：
  - 那些足够老的对象（从次数足够多的GC中幸存下来的）进入终身颐养园；
  - 剩下那些年轻的存活对象进入幸存者乐园里空着的空间。
- 最后，伊甸园和最近腾空的幸存者乐园就可以重用了，因为它们里面已经全是垃圾了。

当伊甸园满了的时候就会触发一次年轻代收集。注意，标记阶段必须遍历整个生存对象图。也就是说如果有个年轻对象被一个终身对象引用了，终身对象所持有的引用也必须被扫描到并标记上。否则只被终身对象引用的伊甸园对象可能会出问题。如果标记阶段不是全遍历，这个伊甸园对象就再也看不到了，而且不可能对它做出正确处理。

3. 完全收集

当年轻代收集不能把对象放进终身颐养园时（空间不够了），就会触发一次完全收集。根据老年代所用的收集器，这可能会牵涉到老年代对象的内部迁移。这样做是为了确保必要时能从老年代对象所占的内存中给大的对象腾出足够的空间。这被称为压缩。

4. 安全点

要想做垃圾收集，至少得让所有应用线程暂停一会儿。但是线程不可能为了GC说停就停。所以它们给执行GC留出了特定的位置——安全点。常见的安全点是方法被调用的地方（“调用点”），不过也有其他安全点。为了执行垃圾收集，所有应用程序线程都必须停在安全点上。

我们暂停一下，先介绍一个简单的工具——`jmap`，它能帮你弄清楚程序运行时的内存使用情况，以及所有内存都用在哪里。我们后续还会介绍一个更先进的GUI工具，但既然很多问题都可以用非常简单的命令解决，你最好应该知道如何使用，而不是直接就使用GUI工具。

6.5.3　jmap

Oracle JVM自带了一些简单的工具，可以帮你了解运行中的进程。jmap是其中最简单的一个，用来显示Java进程的内存映射（它也能分析Java核心文件[①]，甚至能连到远程调试服务器上）。让我们回到电子商务服务器端应用程序的例子上，用jmap对它进行一番探索。

1. 默认视图

jmap最简单的用法是查看连接到进程里的本地类库。除非你的应用程序里有很多JNI代码，否则这种用法通常没什么用，但我们还是会演示一下，以免你忘了指定jmap选项时被它搞糊涂：

```
$ jmap 19306
Attaching to process ID 19306, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11
0x08048000    46K     /usr/local/java/sunjdk/1.6.0_25/bin/java
0x55555000    108K    /lib/ld-2.3.4.so
... some entries omitted
0x563e8000    535K    /lib/libnss_db.so.2.0.0
0x7ed18000    94K     /usr/local/java/sunjdk/1.6.0_25/jre/lib/i386/libnet.so
0x80cf3000    2102K   /usr/local/kerberos/mitkrb5/1.4.4/lib/
     libgss_all.so.3.1
0x80dcf000    1440K   /usr/local/kerberos/mitkrb5/1.4.4/lib/libkrb5.so.3.2
```

一般用得比较多的是-heap和-histo选项，下面我们就来讨论这两个选项。

2. 堆视图

使用-heap选项时，jmap会抓取进程当前的堆快照。在输出结果中能看到构成Java进程堆内存的基本参数。

堆的大小是年轻代、老年代加上PermGen区的总和。但在年轻代内部有伊甸园和幸存者乐园，并且我们还没告诉你这些区域的大小之间有什么关系。这些区域的相对大小是由一个叫做幸存比例的数值决定的。

我们来看一些输出样例。你能在其中看到伊甸园、幸存者乐园（标签为From和To）、终身颐养园（Old Generation）以及一些相关信息：

```
$ jmap -heap 22186
Attaching to process ID 22186, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 20.0-b11

using thread-local object allocation.
Parallel GC with 13 thread(s)
```

---

① Java核心文件（Java core file）主要保存各应用线程在某一时刻的运行位置，即JVM执行到哪个类、哪个方法及哪一行上。它是一个文本文件，打开后可以看到每一个线程的执行栈，以及stack trace的显示。一般Java程序遇到致命问题，在JVM死掉之前会产生两个文件，其中就有Java核心文件，另一个是HeapDump文件。有时为了调试或查找性能问题也会手工生成这两个文件。——译者注

```
Heap Configuration:
   MinHeapFreeRatio = 40
   MaxHeapFreeRatio = 70
   MaxHeapSize      = 536870912 (512.0MB)
   NewSize          = 1048576 (1.0MB)
   MaxNewSize       = 4294901760 (4095.9375MB)
   OldSize          = 4194304 (4.0MB)
   NewRatio         = 2
   SurvivorRatio    = 8
   PermSize         = 16777216 (16.0MB)
   MaxPermSize      = 67108864 (64.0MB)
Heap Usage:
PS Young Generation
Eden Space:
   capacity = 163774464 (156.1875MB)
   used     = 58652576 (55.935455322265625MB)
   free     = 105121888 (100.25204467773438MB)
   35.81301661289516% used
From Space:
   capacity = 7012352 (6.6875MB)
   used     = 4144688 (3.9526824951171875MB)
   free     = 2867664 (2.7348175048828125MB)
   59.10553263726636% used
To Space:
   capacity = 7274496 (6.9375MB)
   used     = 0 (0.0MB)
   free     = 7274496 (6.9375MB)
   0.0% used
PS Old Generation
   capacity = 89522176 (85.375MB)
   used     = 6158272 (5.87298583984375MB)
   free     = 83363904 (79.50201416015625MB)
   6.87904637170571% used
PS Perm Generation
   capacity = 30146560 (28.75MB)
   used     = 30086280 (28.69251251220703MB)
   free     = 60280 (0.05748748779296875MB)
   99.80004352072011% used
```

伊甸园=（From+To）*幸存比例

伊甸园=（From+To）*幸存比例

To空间当前为空

尽管空间的基本构成可能会非常有用，但在这副图里看不到堆里面有什么。如果能看到是哪些对象占用了内存中的空间，你就知道内存都到哪里去了。jmap恰好提供了一个柱状图模式，可以让你看到这些数据的简单统计结果。

3. 柱状视图

柱状视图显示了系统中每个类型的实例（还有一些内部实体）占用的内存量。各个类型按使用内存多少排列，这样就比较容易看到最大的内存猪。

当然，如果所有内存都交给了框架和平台类，这里可能就没你什么事了。但如果真有一个你的类，有了这些信息便能更好地干预它的内存占用。

小小的警告：jmap使用类型内部名称。比如字符数组会写成[C，类对象的数组会显示。

```
$ jmap -histo 22186 | head -30

num     #instances       #bytes  class name
----------------------------------------------
  1:        452779     31712472  [C
  2:         76877     14924304  [B
  3:         20817     12188728  [Ljava.lang.Object;
  4:          2520     10547976  com.company.cache.Cache$AccountInfo
  5:        439499      9145560  java.lang.String
  6:         64466      7519800  [I
  7:         64466      5677912  <constMethodKlass>
  8:         96840      4333424  <methodKlass>
  9:          6990      3384504  <symbolKlass>
 10:          6990      2944272  <constantPoolKlass>
 11:          4991      1855272  <instanceKlassKlass>
 12:         25980      1247040  <constantPoolCacheKlass>
 13:         17250      1209984  java.nio.HeapCharBuffer
 14:         13515      1173568  [Ljava.util.HashMap$Entry;
 15:          9733       778640  java.lang.reflect.Method
 16:         17842       713680  java.nio.HeapByteBuffer
 17:          7433       713568  java.lang.Class
 18:         10771       678664  [S
 19:          1543       489368  <methodDataKlass>
 20:         10620       456136  [[I
 21:         18285       438840  java.util.HashMap$Entry
 22:          9985       399400  java.util.HashMap
 23:         13725       329400  java.util.Hashtable$Entry
 24:          9839       314848  java.util.LinkedHashMap$Entry
 25:          9793       249272  [Ljava.lang.String;
 26:         11927       241192  [Ljava.lang.Class;
 27:          6903       220896  java.lang.ref.SoftReference
```

VM内部对象和类型信息

因为在柱状图模式下输出的数据很多，所以上面只显示了输出内容的一部分。你可能要用grep或其他工具来查看柱状图视图，找到感兴趣的细节。

输出中有很多占用内存的`[C`实体。字符数组数据经常出现在`String`对象里（字符串的内容就存在那里），所以这不奇怪——大多数Java程序里都有很多字符串。但从柱状图中还能看出其他有趣的事情。先来看看下面两个。

前几个实体里唯一一个应用类是`Cache$AccountInfo`——其他全是平台或框架类型——所以它们是开发人员可以完整控制的最重要的类型。`AccountInfo`对象占了很多空间——大概2 500个实体占了10.5 MB（或者每个账号占4 KB）。对于账号细节来说这实在是很多。

这个信息真的非常有用。你已经知道代码里什么占内存最多了。假如老板现在过来告诉你，因为大规模促销，一个月内系统客户数可能要暴增10倍。你知道这可能会给系统增加很多压力——`AccountInfo`对象可是个凶猛的大家伙。虽然你有点担心，但至少你已经开始分析这个问题了。

`jmap`输出的信息可以作为潜在问题处理决策流程的辅助输入。你是不是应该把账号缓存分开，减少该类型保存的信息项，或者买更多的内存给服务器装上。在做出任何决定之前，你还要做很多分析工作，但已经有个起点了。

柱状图模式下还能看到其他有意思的事情，这次指定-histo:live选项。这是告诉jmap只处理存活对象，而不是整个堆（jmap默认会处理所有对象，也包括还没被收集的垃圾）。让我们看看这次输出什么：

```
$ jmap -histo:live 22186 | head -7

 num     #instances         #bytes  class name
----------------------------------------------
   1:          2520       10547976  com.company.cache.Cache$AccountInfo
   2:         32796        4919800  [I
   3:          5392        4237628  [Ljava.lang.Object;
   4:        141491        2187368  [C
```

注意输出的变化——字符数据已经从31MB降到了2MB左右了，证明你第一次看到的String对象里有将近三分之二都是等待回收的垃圾。然而账号对象全是活的，进一步证明了它们是消耗内存的主要力量。

使用jmap时应该稍微谨慎点。进行该操作时JVM还在运行（如果你不走运，还有可能在读取快照期间做了垃圾回收），所以你应该多运行几次，特别是在你看到任何奇怪或太好的结果时。

产生离线导出文件

jmap能创建导出文件，像这样：

```
jmap -dump:live,format=b,file=heap.hprof 19306
```

导出结果可以用来做离线分析，可以留给jmap以后自己用，也可以留给Oracle的jhat（Java堆分析工具）做高级分析。可惜我们没办法在这里全面讨论。

使用jmap可以看到一些基本设置和程序的内存占用。然而要做性能调优，一般需要对GC子系统有更多控制，其标准方式是通过命令行参数，我们来看一些控制JVM的参数，用它们使JVM的行为更适用于你的应用程序。

6.5.4 与GC相关的JVM参数

JVM的参数非常多（最少上百个），用来定制JVM运行时的行为。本节我们会讨论一些跟垃圾收集有关的选项，后续章节中还会讨论其他选项。

非标准的JVM选项

以-X:开头的选项不是标准选项，在其他JVM上可能不可用。

以-XX:开头的是扩展选项，不要随便使用。很多与性能相关的选项都是扩展选项。

有些选项相当于布尔型的参数，并且前面有+或-作为它的开关。还有带参数的选项，比如-XX:CompileThreshold=1000（这个方法会在调用次数达到1000之后才被JIT编译）。还有一些参数（包括很多标准参数）既没有开关也不能带参数。

表6-2中是基本的GC选项，还有这些选项的默认值（如果存在）。

表6-2　基本垃圾收集选项

选　　项	效　　果
-Xms<几MB>m	堆的初始大小（默认2 MB）
-Xmx<几MB>m	堆的最大大小（默认64 MB）
-Xmn<几MB>m	堆中年轻代的大小
-XX:-DisableExplicitGC	让调用System.gc()不产生任何作用

一个常用的小技巧是把-Xms和-Xmx的大小设成一样的。这样进程就会用恰当的堆尺寸运行，没必要在执行过程中调整大小（可能会引发意想不到的降速）。

表中最后一个选项输出GC的标准信息到日志中，我们在下一节会讨论如何解释这些信息。

6.5.5　读懂GC日志

为了充分利用垃圾收集，你需要经常看看子系统在做什么。除了基本的verbose:gc标记，还有很多可以控制输出信息的选项。

别拿GC日志不当回事儿，你可能时不时地就会发现自己被输出信息淹没了。下一节讨论VisualVM时你会发现，有一个可视化工具可以帮你看到VM的行为，这个工具非常有用。不管怎样，会读日志以及了解影响GC的基本选项非常重要，因为有时候你可能没法用GUI工具。最常用的GC日志选项如表6-3所示。

表6-3　用于扩展日志的额外选项

选　　项	效　　果
-XX:+PrintGCDetails	关于GC更详细的细节
-XX:+PrintGCDateStamps	GC操作的时间戳
-XX:+PrintGCApplicationConcurrentTime	在应用线程仍然运行的情况下用在GC上的时间

这些选项组合在一起时，会产生下面这种日志：

```
6.580: [GC [PSYoungGen: 486784K->7667K(499648K)]
1292752K->813636K(1400768K), 0.0244970 secs]
```

我们把它分解，看看每一部分是什么意思：

```
<time>: [GC [<collector name>: <occupancy at start>
➥ -> <occupancy at end>(<total size>)] <full heap occupancy at start>
➥ -> <full heap occupancy at end>(<total heap size>), <pause time> secs
```

第一块是GC的发生时间，从JVM启动开始算，到发生时的秒数。然后是用来收集年轻代的收集器名称（PSYoungGen）。接着是年轻代收集前后占用的内存，以及年轻代的总大小。接着是反映完全收集情况的相同部分。

除了GC日志选项，还有一个选项如果不经解释可能会引起误解。用选项-XX:+PrintGC-ApplicationStoppedTime产生的日志是这样的：

```
Application time: 0.9279047 seconds
Total time for which application threads were stopped: 0.0007529 seconds
Application time: 0.0085059 seconds
Total time for which application threads were stopped: 0.0002074 seconds
Application time: 0.0021318 seconds
```

这些并不一定指GC用了多长时间，而是指在一个从安全点开始的操作中，线程停了多长时间。这包括GC操作，但也包括其他安全点操作（比如Java 6中的偏向锁操作[①]），所以没有十足把握说这是指GC时长。

所有这些信息对记录日志和事后分析都有用，但不容易做可视化处理。而很多开发人员在做初始分析时都喜欢使用GUI工具。好在HotSpot VM（标准的Oracle VM，稍后讨论）自带了一个非常实用的工具。

6.5.6 用VisualVM查看内存使用情况

VisualVM是Oracle JVM自带的可视化工具。它是插件架构，采用标准配置，比JConsole用起来更方便。

图6-6是标准的VisualVM汇总界面。启动VisualVM并把它连接到本地运行的程序上，就能看到这样的界面。（VisualVM也能连接到远程应用上，但有些功能通过网络不可用。）这个界面中VisualVM连接的是MacBook Pro上运行的Eclipse，你可以看到我们用来编写本书代码的Eclipse的设置。图6-6如下所示。

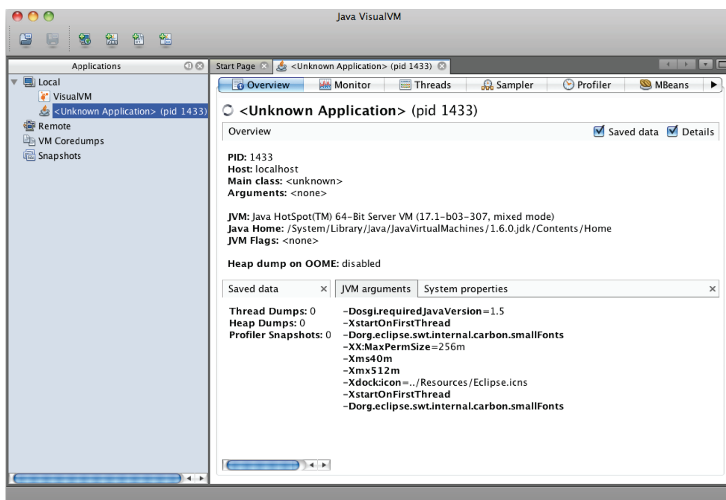


图6-6 VisualVM汇总界面

① 在Java 6之前，加锁会导致一次原子CAS（Compare-And-Set）操作。对于没有争用的资源，该操作会造成无谓的开销。为解决这一问题，Java 6中引入了偏向锁技术，即偏向于第一个加锁的线程，该线程后续加锁操作不需要同步。其基本实现方式为：锁最初为NEUTRAL状态，当第一个线程加锁时，将该锁的状态修改为BIASED，并记录线程ID，这一线程在后续加锁时若发现状态是BIASED并且线程ID是当前线程ID，则只设置一下加锁标志，不需要进行CAS操作。其他线程若要加这个锁，需要使用CAS操作将状态替换为REVOKE，并等待加锁标志清零，以后该锁的状态就变成DEFAULT。这一功能可用`-XX:-UseBiasedLocking`命令禁止。——译者注

右侧面板顶部有很多标签。其中有扩展（Extension）、样例（Sampler）、JConsole、MBeans和VisualVM插件。VisualVM插件为掌握Java运行时的动态情况提供了非常棒的工具。建议你在用VisualVM做任何实际工作前把这些插件都装上。

图6-7展示了内存占用的“锯齿”模式。这绝对是Java平台中内存占用情况的经典表现。它表示对象被分配在伊甸园中，使用，然后在年轻代中被回收。

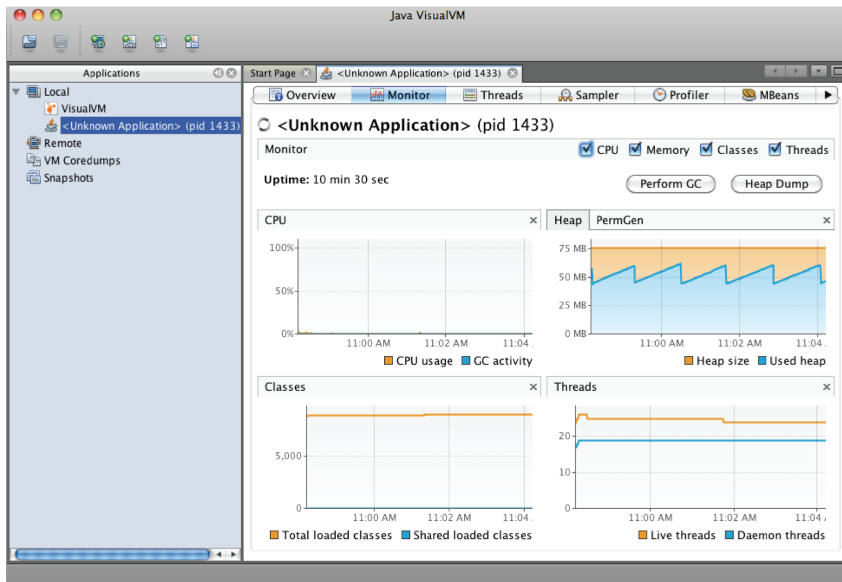


图6-7　VisualVM总览界面

每次年轻代收集之后，被占用的内存量回落到基线水平。这个水平是终身制对象和幸存者对象合起来的用量，可以用它来确定Java进程的健康状况。如果基线在进程工作时保持稳定或者逐渐递减，则表明内存的使用情况非常健康。

如果基线水平上升，也不一定就是出错了，可能只是有些对象的生存期很长，长到足够转入终身颐养园中。在这种情况下，最终会进行一次完全收集。完全收集会导致锯齿模式再次出现，从而使内存占用回落到基线水平。如果完全收集基线持续保持稳定，进程不会耗光内存。

锯齿上斜坡的陡度是进程使用年轻代内存（通常是伊甸园）的频率，这个概念很重要。降低年轻代收集的频率基本上就是降低锯齿的陡度。

内存使用情况的另外一种可视化方式如图6-8所示。你能看到伊甸园、幸存者乐园（S0和S1）、终身颐养园及PermGen区。在程序运行时，你能看到各个空间的大小变化。特别是在年轻代收集之后，可以看到伊甸园变小，幸存者乐园中两个空间的角色也互相转换了。

探索内存系统和运行时环境有助于你理解代码如何运行。相应地，这也表明VM提供的服务对性能影响很大，所以你绝对应该花时间研究一下VisualVM，尤其要结合Xmx和Xms这些选项试一下。

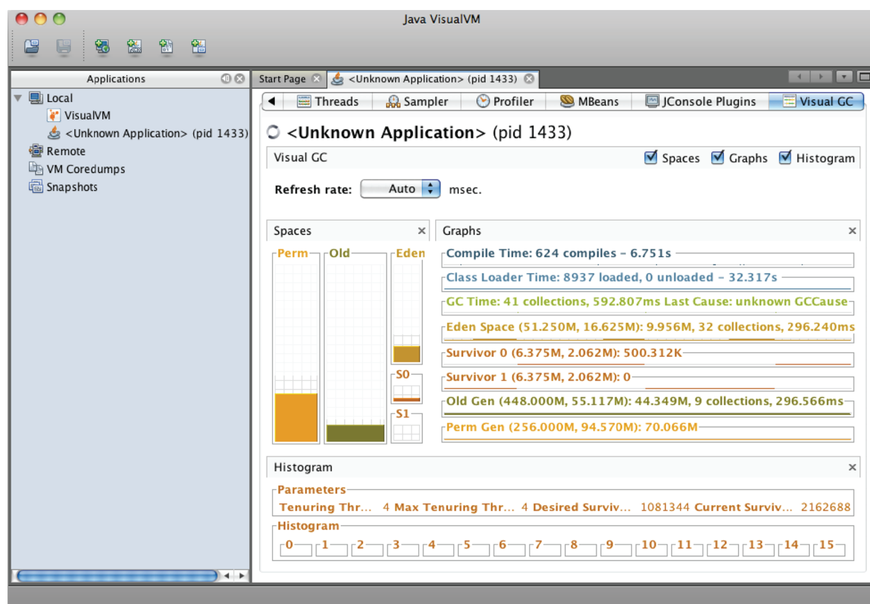


图6-8　VisualVM的可视化GC插件

下一节中，我们将要讨论JVM中的一项新技术，这项技术会在执行过程中自动降低堆内存的占用量。

6.5.7　逸出分析

本节介绍了JVM最近的一项修改，内容仅供参考。程序员不能直接控制或影响这项修改，并且在最近发布的Java中，这项优化是默认的。因此本节中没有太多关于这项修改的信息或例子。所以如果你想了解一下JVM提升自身性能的技巧，请继续。如果没兴趣，可以跳到6.5.8节去研究并发的垃圾收集。

逸出分析乍一看是个相当出人意料的想法。其基本思路是分析方法并确认其中哪个局部变量（的引用类型）只用在方法内部，以及哪些变量不会传入其他方法或从当前方法中返回。

这样JVM就可以在当前方法的栈框架内部创建这个对象，而不再使用堆内存。这会减少程序年轻代收集的次数，从而提高性能。请参见图6-9。

这就是说可以避免堆分配，因为在当前方法返回时，被局部变量占用的内存就自动释放了。用这种不牵扯堆分配的方式分配变量空间不会产生垃圾，当然就不需要收集垃圾。

逸出分析是减少JVM垃圾收集的新办法。它能对线程的年轻代收集次数产生显著影响。经实践证明，它通常能对总体性能产生百分之几的影响。虽然影响不是特别大，但也很有价值，特别是在进程的垃圾收集次数比较多的时候。

从Java 6u23往后，逸出分析是默认打开的，所以新版Java的速度免费提升了。

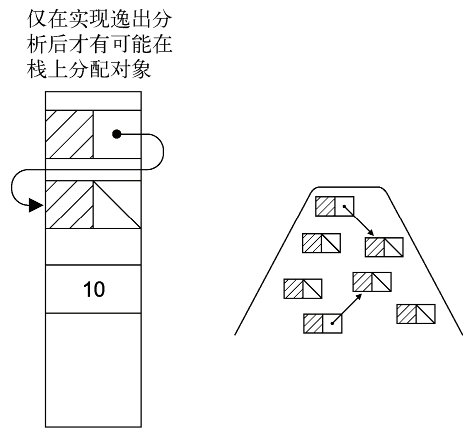


图6-9　逸出分析避免了对象的堆分配

现在我们去看另外一个对代码有巨大影响的环节——收集策略的选择。我们从一个经典的高性能选择（并发标记清除）开始，然后看一看最新的收集器——垃圾优先。

选择高性能收集器有很多原因。应用程序可能会从较短的GC暂停中受益，并且也愿意运行更多线程（占用CPU资源）来加快速度。或者你想控制GC暂停的频度。除了基本的收集器，你还可以用选项迫使平台采用不同的收集策略。在接下来的两节中，我们会介绍两个把这种可能性变成现实的收集器。

6.5.8　并发标记清除

并发标记清除（CMS）收集器是Java 5推荐的高性能收集器，在Java 6中仍然保持了旺盛的生命力。可以通过下面几个选项激活它，如表6-4所示。

表6-4　用于CMS收集器的选项

选　　项	效　　果
`-XX:+UseConcMarkSweepGC`	打开CMS收集
`-XX:+CMSIncrementalMode`	增量模式（一般都需要）
`-XX:+CMSIncrementalPacing`	配合增量模式，根据应用程序的行为自动调整每次执行的垃圾回收任务的幅度（一般都需要）
`-XX:+UseParNewGC`	并发收集年轻代
`-XX:ParallelGCThreads=<N>`	GC使用的线程数

这些选项会覆盖垃圾收集的默认设置，为GC配置有N个并行线程的CMS垃圾收集器。这些线程会尽可能地在并发模式下完成GC工作。

这种并发方式是如何工作的呢？下面是与标记清除相关的三个重要事实：

- 某种世界停转（简称STW）的暂停是不可避免的；
- GC子系统绝对不能漏掉存活对象，这样做会导致JVM垮掉（或者更糟）；

❑ 只有所有应用线程都为整体收集暂停下来，才能保证收集所有的垃圾。

CMS利用了最后一点。它制造两个非常短暂的STW暂停，并且在GC周期的剩余时间和应用程序的线程一起运行。这表明它愿意跟“伪阴性”妥协，由于竞争危害而无法标识某些垃圾（被漏掉的垃圾会在下一个GC周期中得到收集）。

CMS还要在运行时做复杂的记账工作，记录哪些是垃圾，哪些不是。这些额外的开销是为了在不停止应用线程的情况下运行GC所付出的代价。CMS在有更多CPU核心的机器上会表现得更好，并且会制造更频繁的短暂暂停。它的日志输出如下所示：

```
2010-11-17T15:47:45.692+0000: 90434.570: [GC 90434.570:
[ParNew: 14777K->14777K(14784K), 0.0000595 secs]90434.570:
[CMS: 114688K->114688K(114688K), 0.9083496 secs] 129465K->117349K(129472K),
[CMS Perm : 49636K->49634K(65536K)] icms_dc=100 , 0.9086004 secs]
[Times: user=0.91 sys=0.00, real=0.91 secs]
```

这些日志和6.4.4节中基本的GC日志差不多，但增加了CMS和CMS Perm收集器部分。

最近几年，CMS作为最佳高性能收集器的地位受到了挑战，挑战者是垃圾优先（G1）收集器。我们来看看这颗冉冉升起的新星，了解一下它的新颖方法，以及它能够突破所有现存的Java收集器的原因。

6.5.9 新的收集器：G1

G1是Java平台中崭新的收集器。本来想把它和Java 7一起发布，但后来作为预发布版本跟Java 6一起发布了，到Java 7时就是成品了。它在Java 6中并没有得到广泛的应用，但随着Java 7逐渐普及，有望让G1成为高性能应用（也可能是所有应用）的默认选择。

G1的核心思想是暂停目标（pause goal），也就是程序在执行时能为GC暂停多长时间（比如每5分钟20ms）。G1会竭尽所能达成暂停目标。它和我们原来遇到的收集器完全不同，并且开发人员对GC如何执行有更多控制权。

G1不是真正的分代式垃圾收集器（尽管它仍然使用标记清除法）。相反，G1把堆分成大小相同的区域（比如每个1 MB），不区分年轻区和年老区。暂停时，对象被撤到其他区域（就像伊甸园对象被挪到幸存者乐园一样），清空的区域被放回到（空白区的）自由列表上。这种将堆划分为大小相同区域的做法如图6-10所示。

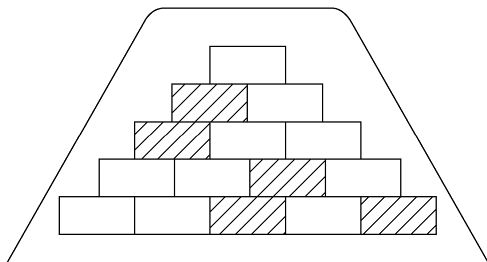


图6-10 G1如何划分堆空间

这个新的收集策略让Java平台可以统计收集单个区域需用的平均时长。这样你就可以在合理范围内指定一个暂停目标。G1只会在有限的时间内收集尽可能多的区域（尽管在收集最后一个区域时所用的时间可能比预期的长）。

要打开G1，需要用到表6-5中的选项。

表6-5　G1收集器的选项

选　　项	效　　果
`-XX:+UseG1GC`	打开G1收集
`-XX:MaxGCPauseMillis=50`	告诉G1它在一次收集中暂停的时间应该尽量保持在50ms以内
`-XX:GCPauseIntervalMillis=200`	告诉G1它将两次收集的时间间隔尽量保持在200ms以上

这些选项可以组合，比如设置最大暂停目标是50 ms，暂停间隔不能少于200 ms。当然，GC系统所能承受的压力是有限的。必须有充足的暂停时间把垃圾取出来。每隔100年1ms的暂停目标肯定是不现实的。

G1可以支持的负载和应用类型范围很广。如果你的应用程序已经到了需要对GC调优的地步，G1会是一个不错的选择。

在下一节中，我们会介绍JIT编译。对于很多或大多数程序来说，这是唯一一个可以为产生高性能代码做出最大贡献的因素。我们会学习JIT编译的基础知识，最后解释一下如何打开JIT编译的日志，让你能够判断正在编译哪个方法。

6.6　HotSpot 的 JIT 编译

正如我们在第1章所讲，Java是一种“动态编译”语言。也就是说在程序运行时，其中的类还会再进行一次编译，然后转换成机器码。

这个过程称为即时编译或JITing，并且通常是一次处理一个方法。要在庞大的代码库中找出其中的重要部分，理解这个过程是关键。

下面是一些与JIT编译有关的基本事实。

- 几乎所有现代JVM中都有某种JIT编译器。
- 相比较而言，纯粹解释型的VM要慢得多。
- 编译过的方法在运行速度上要比解释型的代码快很多，非常多。
- 先编译用得最多的方法，这是有道理的。
- 在做JIT编译时，先处理唾手可得的编译很重要。

按照最后一点，我们应该先研究编译过的代码，因为在正常情况下，所有仍然处于解释状态下的方法都没有已经编译过的方法运行频繁。偶尔会有无法编译的方法，但非常罕见。

方法一开始都是以字节码形态存在的，有调用时JVM只会对字节码进行解释并执行，同时记录方法被调用的次数及其他一些统计数据。当被调用次数达到某个阈值（默认10 000次）后，如

果它是合格的方法，就会有个JVM线程在后台把它的字节码编译成机器码。如果编译成功，以后所有对该方法的调用都会用它的编译结果，除非出现了某些导致检验失效的情况，或者出现了逆优化[1]。

根据实际情况，方法编译后产生的机器码运行速度可能比解释模式下的字节码快100倍。改善性能通常都要先弄明白程序中哪些方法比较重要，以及哪些重要的方法被编译了。

为什么要动态编译？

有时人们会问，Java平台为什么要费心去做动态编译——为什么不提前编译好（像C++一样）。第一个答案通常都是：因为用平台无关的东西（.jar和.class文件）作为基本部署单位要比为每个目标平台做一份不同的编译好的二进制文件更轻松。

另外一种答案是动态编译会给编译器提供更多信息。具体地说，提前（AOT）编译的语言得不到运行时的任何信息——比如某个指令是否可用，其他的硬件细节以及代码运行情况的统计数据。这些变数让事情变得很有趣，使得Java这样的动态编译语言实际上可能会比提前编译的语言运行得更快。

在接下来对JITing机制的讨论中，我们所说的JVM特指HotSpot。后续讨论中很多通用内容也适用于其他VM，但在具体细节上可能会有很大出入。

我们会先介绍一下HotSpot提供的几个JIT编译器，然后解释HotSpot中最有力的两项优化技术（内联和独占派发）。在本节的结尾，我们会告诉你如何打开方法编译日志，以便你可以看到被编译的确切方法。下面有请HotSpot。

6.6.1 介绍HotSpot

Oracle收购Sun时拿到了HotSpot VM（原来收购BEA时还拿到一个JRockit）。HotSpot是OpenJDK的基础。它有两种运行模式——客户端模式和服务器端模式。可以在启动JVM时指定`-client`或`-server`选项来选择不同的模式。（必须是命令行中的第一个选项。）每种模式都有各自适用的应用程序。

1. 客户端编译器

客户端编译器主要用于GUI应用程序。在这个领域中，操作的一致性至关重要，所以客户端编译器（有时叫C1）在编译时所做的决定往往更保守。也就是说它不能因为要取消一个经证实不正确或基于错误假设的优化决定而意外暂停。

2. 服务器端编译器

相反，服务器端编译器（C2）在编译时会大胆假设。为了确保代码正确运行，C2会快速地

[1] JVM的动态优化技术可能会基于一些大胆（甚至不安全）的假设来编译字节码。比如假定要处理的数据都属于某一类，而在编译结果中只保留处理该类数据的程序分支。如果假设不成立，则JVM只能放弃编译结果，回去解释并执行原来的字节码，这一过程被称为逆优化。——译者注

做一次运行时检查（通常被称为警戒条件），以确保假设有效。如果假设无效，它会取消这次编译，并尝试别的编译。这种大胆假设的方式比保守的客户端编译器产生的编译结果性能好很多。

3. 实时Java

近年来出现了一种实时Java平台，有些开发人员好奇为什么那些需要表现出高性能的代码不直接用这个平台（它是独立的JVM，不是HotSpot选件）。那是因为实时系统不一定是最快的。

实时编程的关注点实际上是承诺能否兑现。从统计角度讲，实时系统是为了让执行操作的时间尽量保持一致，并且为了达成这个目的，它可能会牺牲一些平均等待时间。为了让运行状况保持一致，整体性能是可以受到轻微影响的。

图6-11中有两组代表等待时间的点阵。系列2（上面那组点阵）的平均等待时间在增长（因为它的等待时间刻度更高），但方差在减小，因为这些点比系列1中的点更靠近自己的平均值，系列1的点阵相较而言分布更加广泛。

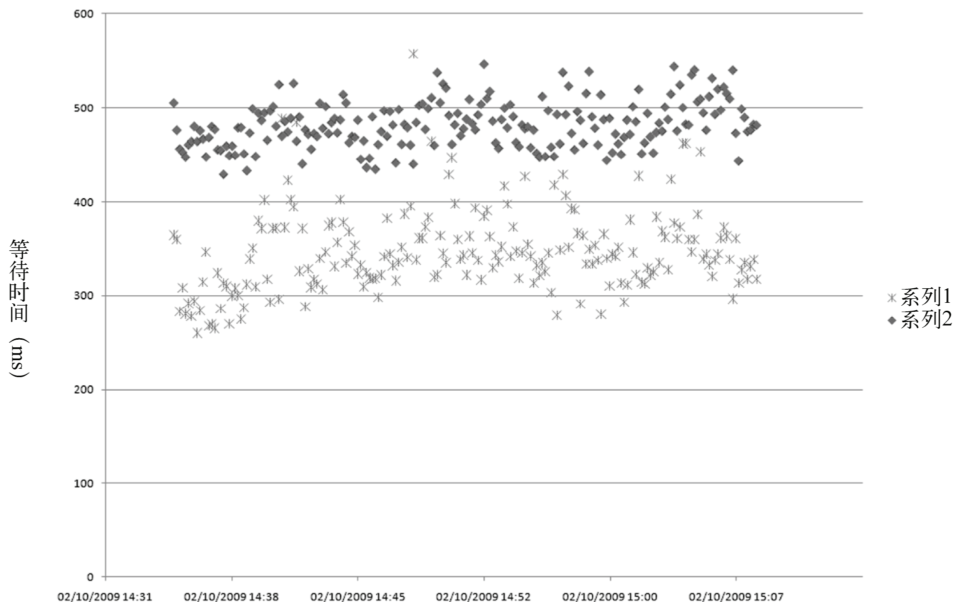


图6-11　方差和均值的变化

但希望实现高性能表现的团队想要的是更低的平均等待时间，即便以更高的方差为代价，所以他们通常会选择服务器端编译器的大胆优化策略（对应系列1）。

接下来我们会讨论所有运行时（服务器端、客户端和实时）广泛采用的技术，这项技术使它们表现得更好。

6.6.2　内联方法

内联是HotSpot的最大卖点之一。内联的方法不再是被调用，而是将调用方法的代码直接放

到调用者内部。

平台有这方面的优势，编译器可以根据运行时的统计数据（方法的调用频率）和其他因素（比如会不会因为调用者方法太多而对代码缓存产生影响）来决定如何处理内联。也就是说HotSpot编译器所做的内联决策比提前编译的编译器更智能。

方法的内联是完全自动的，并且默认参数值几乎适用于任何情况。但也有选项可以用来控制内联方法大小，以及方法在成为内联候选之前的调用频率要达到多高。对于好奇的程序员来说，这些选项对于深入了解内联如何工作很有帮助。通常它们对于生产环境下的代码用处不大，并且应该作为性能调优的最后选择，因为它们对运行时系统的性能可能存在不可预测的影响。

访问器方法怎么处理？

有些开发人员错误地认为访问器方法（访问私有变量的公共方法）不能由HotSpot内联。他们认为变量是私有的，方法调用不能因为优化而去掉，不能在类外访问这个变量。这种想法不对。HotSpot把方法编译成机器码时能够并且会忽略访问控制，不用访问器方法直接访问私有域。这并不违背Java的安全模型，因为所有访问控制都在类加载和连接阶段检查过。

如果你还不信，可以做个练习，写一个跟代码清单6-2类似的测试类，对比一下预热过的访问器方法的速度和直接访问公共域的速度。

6.6.3 动态编译和独占调用

独占调用就是这种大胆优化的例子之一。它是基于大量观察做出的优化，像下面这种对象上的方法调用：

```
MyActualClassNotInterface obj = getInstance();
obj.callMyMethod();
```

只会在一种类型的对象上调用。换句话说，就是调用点`obj.callMyMethod()`几乎不会同时碰到一个类和它的子类。这时可以把Java方法查找替换为`callMyMethod()`编译结果的直接调用。

提示 独占派发提供了一个剖析JVM运行时的例子，允许Java平台进行C++这种AOT语言实现不了的优化。

出于非技术的原因，`getInstance()`方法有时不能返回`MyActualClassNotInterface`类型的对象，而其他情况下不能返回一些子类的对象，但实际上这种情况几乎从没发生过。但为了防止这种情况出现，会有一个运行时检查来确保对象的类型是由编译器按预期插入的。如果这个预期被违背，运行时会取消优化，程序甚至都不会注意也不会犯任何错误。

只有服务器端编译器才会做这种大胆的优化。实时和客户端编译器都不会这样做。

6.6.4　读懂编译日志

我们来看一个例子，了解一下如何使用JIT编译日志。依巴谷星表中详细列出了从地球上可以观测到的星星。我们的程序会处理这个目录，产生能在指定夜晚、指定地址看到的星图。

我们来看这个程序输出的一些日志，看看在星图应用运行时编译了哪些方法。我们用的关键选项是-XX:+PrintCompilation。我们前面简单讨论过这个扩展选项。把这个选项加到启动JVM的命令里是告诉JIT编译线程输出标准日志。这些日志表明方法超过编译阈值并被转成机器码的时间。

```
   1       java.lang.String::hashCode (64 bytes)
   2       java.math.BigInteger::mulAdd (81 bytes)
   3       java.math.BigInteger::multiplyToLen (219 bytes)
   4       java.math.BigInteger::addOne (77 bytes)
   5       java.math.BigInteger::squareToLen (172 bytes)
   6       java.math.BigInteger::primitiveLeftShift (79 bytes)
   7       java.math.BigInteger::montReduce (99 bytes)
   8       sun.security.provider.SHA::implCompress (491 bytes)
   9       java.lang.String::charAt (33 bytes)
   1% !    sun.nio.cs.SingleByteDecoder::decodeArrayLoop @ 129 (308 bytes)
...
  39       sun.misc.FloatingDecimal::doubleValue (1289 bytes)
  40       org.camelot.hipparcos.DelimitedLine::getNextString (5 bytes)
  41  !    org.camelot.hipparcos.Star::parseStar (301 bytes)
...
   2% !    org.camelot.CamelotStarter::populateStarStore @ 25 (106 bytes)
  65 s     java.lang.StringBuffer::append (8 bytes)
```

这是非常典型的PrintCompilation输出。这些日志表明了“热”到可以编译的方法。跟你想的一样，第一个被编译的方法很可能是平台方法（比如String#hashCode）。再过一段时间，应用方法（比如org.camelot.hipparcos.Star#parseStar方法，在例子中用于分析天文目录里的记录）也会被编译。

这些输出中每行都有个数字，表明了这些方法在这次运行中的编译顺序。注意，由于平台的动态性质，这个顺序在每次运行时可能会稍有变化。这里还有一些其他域。

- s——表明该方法是同步的。
- !——表明方法有异常处理。
- %——当前栈替换（OSR）。这个方法被编译了，并且换掉了运行代码中的解释型版本。注意，OSR方法有它们自己的计数方案，从1开始。

小心僵尸

当查看用服务器端编译器（C2）运行代码的样例日志时，你可能偶尔会看到“变得无法进入”和“变成僵尸”这样的字眼。这表明由于类加载操作（通常情况下），某个已经被编译过的特定方法现在无效了。

逆优化

如果经证实代码优化所基于的假设是不真实的，HotSpot可以对代码进行逆优化操作。在许多情况下，它会重新考虑，尝试不同的优化。因此同一个方法可能会被逆优化和重编译几次。

过了一段时间，你会看到被编译的方法数量趋于稳定。编译好的代码达到了一个稳定的状态，并且大多数代码会保持不变。哪些方法被编译取决于所用的JVM版本和OS平台。并不是所有平台都会产生相同的编译方法集合，并且给定方法编译代码的大小也不会完全一样。就像性能调优里很多其他东西一样，这也应该进行测量，并且结果可能会让人大吃一惊。即便看上去相当简单的Java方法，在Solaris和Linux上经JIT编译生成的机器码也会有五分之一的差异。测量是必不可少的。

6.7　小结

性能调优不是盯着你的代码期待奇迹，或者给代码喝一罐快速修复药水。相反，性能调优需要细致测量，关注细节，还需要你的耐心。你要不断减少测试中出现的错误源，直到引发性能问题的真正凶手出现。

我们先来看看在JVM动态环境中进行性能调优的要点。

- JVM是极为强大的复杂运行时环境。
- JVM的性质使得有时候优化其中的代码很有挑战性。
- 你必须通过测量准确地找到问题的真正所在。
- 要特别注意垃圾收集子系统和JIT编译器。
- 监测还有其他一些工具对你真的很有帮助。
- 学会阅读日志和平台的其他指标——有时不能使用工具。
- 你必须测量并设置目标（这个太重要了，所以我们要一再提起）。

现在你应该具备探索和实验Java平台的高级性能特性所需的基础知识了，并且能够理解性能机制如何影响你的代码。希望你能开放心态，以足够的信心和经验去分析这些数据，并能把这种见解应用于你自己的性能问题。

我们会在下一章看到JVM上除Java语言之外的其他语言，平台的很多性能特性适用范围非常广泛——特别是JIT编译器和GC的相关知识。

Part 3

第三部分

JVM 上的多语言编程

这一部分专门探索 JVM 上的新语言范式和多语言编程。

JVM是一个迷人的运行时环境：它提供的不仅是性能和能力，还赋予了程序员惊人的灵活性。实际上，JVM 是探索 Java 之外的语言的关口，并且会让你尝试一些不同的编程方式。

如果你只用Java写过程序，可能想知道学习其他语言会有什么好处。就像我们在第1章说的，成为优秀 Java 开发人员的本质就是对 Java 语言、平台和生态系统的方方面面掌握得越来越全面。这包括能够欣赏那些目前刚刚起步，但不久的将来就会变得不可或缺的主题。

未来已经发生，只是分布尚不均匀。

——威廉·吉布森

事实证明，很多未来需要的新想法已经出现在函数式编程等其他 JVM 语言中了。学习新 JVM 语言的过程中，我们可以一瞥另一个世界，我们未来的某些项目很可能就跟它很像。从不同的视角探索问题能帮我们重新审视已有的知识。学习新语言可以开启新的可能性，我们可能会发现自己不知道的新天赋，掌握新技能，而这些东西总有一天会派上用场。

第 7 章会解释一下为什么 Java 不是解决所有问题的理想语言、为什么函数式编程概念有用，以及如何为特定项目选择一种非 Java 语言。

最近，很多书和博客里都提出一种观点，认为函数式编程很快就会成为每个开发人员职业生涯中的重要角色。很多文章都把函数式编程描述得令人生畏，却常常讲不清楚函数式编程怎么在 Java 这样的语言中“发光发热”。

实际上，函数式编程根本算不上一个整体结构。相反，它更像一种风格，开发人员思考方式上的一个过渡。第 8 章会给出一个用 Groovy 语言编写的、稍微带点儿函数式编程味道的例子，就是用一种更清晰的、不太容易出 bug 的风格来处理集合的代码。在第 9 章，我们会用 Scala 语言讨论对象—函数式风格。第 10 章会用 Clojure 语言看一下纯粹的函数式编程（它甚至超过了面向对象）方式。

在第四部分，我们会介绍几个真实案例，针对这些案例，其他语言能够给出更好的解决方案。如果你不信，可以提前看一下第四部分，然后再回来学习应用那些技术所需的语言。

第 7 章

备选JVM语言

本章内容
- 为什么应该使用备选JVM语言
- 语言的类型
- 备选语言的选择标准
- JVM如何处理备选语言

如果你用Java做过大项目，可能已经注意到了，Java有时稍显繁琐和笨拙。你甚至可能希望它不是这样的——总之要再容易点儿。

好在JVM很棒！实际上，它太棒了，Java以外的其他语言也可以很自然地把它当成栖息地。我们在本章里会告诉你为什么要把其他JVM编程语言加入到我们的项目中，以及如何做到这一点。

我们会讨论描述不同语言类型（比如静态与动态）的方式、为什么用备选语言，以及选择它们时有哪些标准。我们还会介绍三种语言：Groovy、Scala和Clojure，并在第三部分和第四部分中更深入地探讨它们。

然而在开始之前，你需要对Java的缺点有更清楚的认识。下一节有一个扩展示例，它突出了Java语言中一些恼人的地方，指出了它未来的发展方向为*函数式编程*风格。

7.1 Java 太笨？纯粹诽谤

假设你要在一个交易（事务）处理系统中编写一个新组件。这个系统的简化视图如图7-1所示。

在图中可以看到，系统有两个数据源：上游的收单系统（可以通过Web服务查询）和下游的派发数据库。

这是一个很现实的系统，是Java开发人员经常构建的系统。我们在这一节里准备引入一小段代码把两个数据源整合起来。你会看到Java解决这个问题有点笨拙。之后我们会介绍函数式编程的一个核心概念，并展示一下怎么用*映射*（map）和*过滤器*（filter）等函数式特性简化很多常见的编程任务。你会看到Java由于缺乏对这些特性的直接支持，编程会困难不少。

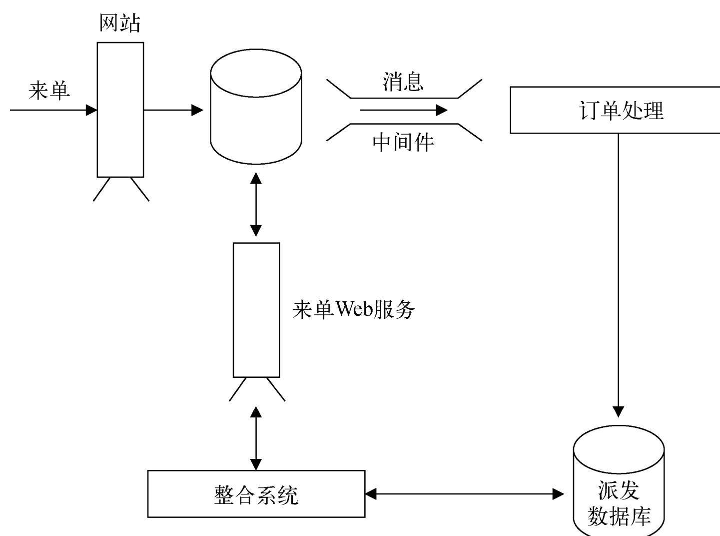


图7-1 交易处理系统的例子

7.1.1 整合系统

我们需要一个整合系统来检查数据确实到了数据库。这个系统的核心是reconcile()方法，它有两个参数：sourceData（来自于Web服务的数据，归结到一个Map中）和dbIds。

你需要从sourceData中取出main_ref键值，用它跟数据库记录的主键比较。代码清单7-1是进行比较的代码。

代码清单7-1 整合两个数据源

```
public void reconcile(List<Map<String, String>> sourceData,
Set<String> dbIds) {
  Set<String> seen = new HashSet <String>();
  MAIN: for (Map<String, String> row : sourceData) {
    String pTradeRef = row.get("main_ref");          // 假定pTradeRef永远不会为null

    if (dbIds.contains(pTradeRef)) {
      System.out.println(pTradeRef +" OK");
    seen.add(pTradeRef);
  } else {
    System.out.println("main_ref: "+ pTradeRef +" not present in DB");
  }
  }

  for (String tid : dbIds) {                        // 特殊情况
    if (!seen.contains(tid)) {
      System.out.println("main_ref: "+ tid +" seen in DB but not Source");
    }
  }
}
```

这里主要是检查收单系统中的所有订单是否都出现在派发数据库里。这项检查由打上了MAIN标签的for循环来做。

还有另外一种可能。比如有个实习生通过管理界面做了些测试订单（他没意识到这些订单用的是生产系统）。这样订单数据会出现在派发数据库里，但不会出现在收单系统中。

为了处理这种特殊情况，还需要一个循环。这个循环要检查所见到的集合（同时出现在两个系统中的交易）是否包含了数据库中的全部记录。它还会确认那些遗漏项。下面是这个样例的一部分输出：

```
7172329 OK
1R6GV OK
1R6GW OK
main_ref: 1R6H2 not present in DB
main_ref: 1R6H3 not present in DB
1R6H6 OK
```

哪儿出错了？原来是上游系统不区分大小写而下游系统区分，在派发数据库里表示为1R6H12的记录实际上是1r6h2。

如果你检查一下代码清单7-1，就会发现问题出在contains()方法上。contains()方法会检查其参数是否出现在目标集合中，只有完全匹配时才会返回true。

也就是说其实你应该用containsCaseInsensitive()方法，可这是一个根本就不存在的方法！所以你必须把下面这段代码

```
if (dbIds.contains(pTradeRef)) {
  System.out.println(pTradeRef +" OK");
  seen.add(pTradeRef);
} else {
  System.out.println("main_ref: "+ pTradeRef +" not present in DB");
}
```

换成这样的循环：

```
for (String id : dbIds) {
  if (id.equalsIgnoreCase(pTradeRef)) {
    System.out.println(pTradeRef +" OK");
    seen.add(pTradeRef);
    continue MAIN;
  }
}
System.out.println("main_ref: "+ pTradeRef +" not present in DB");
```

这看起来比较笨重。只能在集合上执行循环操作，不能把它当成一个整体来处理。代码既不简洁，又似乎很脆弱。

随着应用程序逐渐变大，简洁会变得越来越重要——为了节约脑力，你需要简洁的代码。

7.1.2　函数式编程的基本原理

希望上面的例子中的两个观点引起了你的注意。

- 将集合作为一个整体处理要比循环遍历集合中的内容更简洁，通常也会更好。
- 如果能在对象的现有方法上加一点点逻辑来调整它的行为是不是很棒呢？

如果你遇到过那种基本就是你需要，但又稍微差点儿意思的集合处理方法，你就明白不得不再写一个方法是多么沮丧了，而函数式编程（FP）恰好搔到了这个痒处。

换种说法，简洁（并且安全）的面向对象代码的主要限制就是，不能在现有方法上添加额外的逻辑。这将我们引向了FP的大思路：假定确实有办法向方法中添加自己的代码来调整它的功能。

这意味着什么？要在已经固定的代码中添加自己的处理逻辑，就需要把代码块作为参数传到方法中。下面这种代码才是我们真正想要的（为了突出，我们把这个特殊的contains()方法加粗了）：

```
if (dbIds.contains(pTradeRef, matchFunction)) {
  System.out.println(pTradeRef +" OK");
  seen.add(pTradeRef);
} else {
  System.out.println("main_ref: "+ pTradeRef +" not present in DB");
}
```

如果能这样写，contains()方法就能做任何检查，比如匹配区分大小写。这需要能把匹配函数表示成值，即能把一段代码写成“函数字面值”并赋值给一个变量。

函数式编程要把逻辑（一般是方法）表示成值。这是FP的核心思想，我们还会再次讨论，先看一个带点儿FP思想的Java例子。

7.1.3 映射与过滤器

我们把例子稍微展开一些，并放在调用reconcile()的上下文中：

```
reconcile(sourceData, new HashSet<String>(extractPrimaryKeys(dbInfos)));

private List<String> extractPrimaryKeys(List<DBInfo> dbInfos) {
  List<String> out = new ArrayList<>();
  for (DBInfo tinfo : dbInfos) {
    out.add(tinfo.primary_key);
  }

  return out;
}
```

extractPrimaryKeys()方法返回从数据库对象中取出的主键值（字符串）列表。FP粉管这叫map()表达式：extractPrimaryKeys()方法按顺序处理List中的每个元素，然后再返回一个List。上面的代码构建并返回了一个新列表。

注意，返回的List中元素的类型（String）可能跟输入的List中元素的类型（DBInfo）不同，并且原始列表不会受到任何影响。

这就是“函数式编程”名称的由来，函数的行为跟数学函数一样。函数f(x)=x*x不会改变输入值2，只会返回一个不同的值4。

便宜的优化技巧

调用reconcile()时，有个实用但小有难度的技巧：把extractPrimaryKeys()返回的List传入HashSet构造方法中，变成Set。这样可以去掉List中的重复元素，reconcile()方法调用的contains()可以少做一些工作。

map()是经典的FP惯用语。它经常和另一个知名模式成对出现：filter()形态，请看代码清单7-2。

代码清单7-2　过滤器形态

```
List<Map<String, String>> filterCancels(List<Map<String, String>> in) {
  List<Map<String, String>> out = new ArrayList<>();    <-- 防御性复制
  for (Map<String, String> msg : in) {
    if (!msg.get("status").equalsIgnoreCase("CANCELLED")) {
      out.add(msg);
    }
  }

  return out;
}
```

注意其中的防御性复制，它的意思是我们返回了一个新的List实例。这段代码没有修改原有的List（filter()的行为跟数学函数一样）。它用一个函数测试每个元素，根据函数返回的boolean值构建新的List。如果测试结果为true，就把这个元素添加到输出List中。

为了使用过滤器，还需要一个函数来判断是否应该把某个元素包括在内。你可以把它想象成一个向每个元素提问问题的函数："我应该允许你通过过滤器吗？"

这种函数叫做*谓词函数*（predicate function）。这里有一个用伪代码（几乎就是Scala）编写的方法：

```
(msg) -> { !msg.get("status").equalsIgnoreCase("CANCELLED") }
```

这个函数接受一个参数（msg）并返回boolean值。如果msg被取消了，它会返回false，否则返回true。用在过滤器中时，它会过滤掉所有被取消的消息。

这就是你想要的。在调用整合代码之前，你需要移除所有被取消的订单，因为被取消的订单不会出现在派发数据库中。

事实上，Java 8准备采用这种写法（受到了Scala和C#语法的强烈影响）。我们在第14章还会讨论这个主题，但在那之前我们会遇到几次函数字面值（也称为lambda表达式）。

我们接着往下看，讨论一下其他情况，从JVM上可用的语言类型开始（有时候我们也把这称为*语言生态学*）。

7.2　语言生态学

编程语言有多种不同的流派和类型。也就是说，不同的语言有不同的编码风格和编码方式。如果想掌握这些风格并让它们为你所用，你得弄明白这些差异以及如何给语言分类。

注意　这些分类可以帮助思考语言的多样性。尽管某些分法可能更清晰，但没有哪种是完美的。

最近几年，语言在添加新特性时有跨越各种分类的趋势。这就是说，你最好认为某种语言比其他语言"函数化程度更低"，或者"虽然是动态类型，但必要时也有可选的静态类型"。

我们将要讨论的分类是“解释型与编译型”、“动态与静态”、“命令式与函数式”，还有在JVM上重新实现的语言与原生语言。通常这些分类用来明确语言的边界，不要用学院化方式把它们当做完整精确的分类。

Java是运行时编译、静态类型的命令式语言。它强调安全性、代码清晰、性能，并乐于表现出一定程度的繁琐和死板（比如在部署中）。不同的语言可能侧重不同，比如动态类型的语言可能更看重部署速度。

我们先从解释型与编译型分类开始介绍。

7.2.1 解释型与编译型语言

解释型语言是那种源码是什么就执行什么的语言，不会在执行开始之前把整个程序转成机器码。编译型语言则不同，一开始就要用编译器把人类可读的源码变成二进制形式。

这种分别最近变模糊了。80年代和90年代早期这两类语言的边界还相当清晰：C/C++及类似的语言是编译型，Perl和Python是解释型。但Java同时兼具编译型和解释型两种特性，这一点我们已经在第1章讲过了。字节码的出现使这个界限更模糊了。人类肯定读不了字节码，但它也不是真正的机器码。

对于本书中要研究的JVM语言，我们划分的边界是该语言是否会将源码编译为类文件并且执行。不产生类文件的语言会由解释器（可能是用Java写的）逐行执行源码。有些语言既有编译器也有解释器，还有些既有解释器又有产生JVM字节码的即时编译器（JIT）。

7.2.2 动态与静态类型

在动态类型语言中，变量在不同时间可能会有不同的类型。我们以一小段简单的JavaScript代码为例，JavaScript是著名的动态语言。即便你不了解这种语言，也应该很容易理解下面的代码：

```
var answer = 40;
answer = answer + 2;
answer = "What is the answer? " + answer;
```

在这段代码中，变量answer一开始被赋值为40，当然，是个数值。然后给它加上2，变成了42。之后我们给answer赋了个字符串值。这在动态语言中是非常普遍的技术，不会引起语法错误。

JavaScript解释器也能分清两种+操作符的用法。第一个+是数字相加——把2加到40上，而在下一行中，解释器能从上下文中推导出开发人员要做字符串合并。

注意 这里的关键是动态类型语言跟踪变量值的类型（比如数字或字符串）信息，而静态类型语言跟踪变量的类型信息。

静态类型非常适合编译型语言，因为所有类型信息都在变量上，跟变量的值没有关系。这样很容易在编译时推导潜在的类型系统违规行为。

动态类型语言把类型信息放在变量所持有的值上。也就是说很难提前发现类型违规行为，因为推导所需的信息直到运行时才能得到。

7.2.3 命令式与函数式语言

Java 7是典型的命令式语言。命令式语言把程序的运行状态建模为可修改的数据，用一系列指令来改变运行状态。因此，在命令式语言中，程序状态才是核心概念。

命令式语言主要分为两类。一种是过程语言，比如BASIC和FORTRAN。这种语言把代码和数据完全分开，有简单的代码操作数据范式。另外一种是面向对象（OO）语言，数据和代码（以方法的形式）共同封装在对象中。面向对象语言中或多或少地存在元数据（比如类信息）引入的额外结构。

函数式语言不同，它把计算本身当做最重要的概念。函数式语言跟过程语言一样对值进行操作，但它不会修改输入，而是像数学函数一样返回新值。

如图7-2所示，函数被看做“小处理机”，输入值并输出新值。它们没有任何自己的状态，并且把它们和任何外部状态绑在一起也没有任何意义。这就是说一切皆对象的世界观跟函数式语言的自然观点有些分歧。

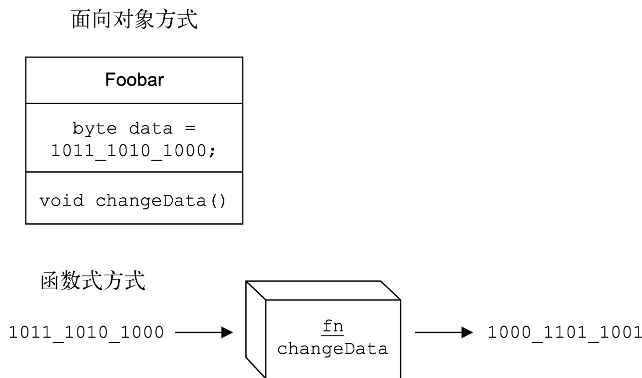


图7-2 命令式和函数式语言

在接下来的三章里，每章重点介绍一种语言，并且都会以前面对函数式编程的讨论为基础。我们会从Groovy开始，它带“一点儿函数式风格”，用我们在7.1节讨论过的方式处理集合；然后是Scala，对FP的利用更加充分；最后是Clojure（纯粹的函数式语言，完全没有面向对象特性）。

7.2.4 重新实现的语言与原生语言

JVM语言之间的另一个重要区别是重新实现已有语言与专门以JVM为目标的划分。通常来说，那些专门以JVM为目标写的语言能把自己的类型系统跟JVM的原生类型结合得更紧密。

下面是三种重新实现已有语言的JVM语言。

- JRuby在JVM上重新实现了Ruby语言。Ruby是一个动态类型的面向对象语言，有些函数式特性。它在JVM上基本算解释型的，但最近发布的版本中有一个运行时JIT编译器，在适当条件下可以生成JVM字节码。
- Jython由Jim Hugunin在1997年发起，当时是为了在Python中使用高性能的Java类库。它在JVM上重新实现了Python，因此是动态的，总体还算面向对象语言。它的运作方式是先在内部生成Python字节码，然后再转换成JVM字节码。这使它能在Python典型的解释模式（看起来像）下工作。通过生成JVM字节码，并把结果类文件保存到硬盘上，它也能在预先（AOT）编译模式下工作。
- Rhino最初是由Netscape开发的，后来转给Mozilla项目。它在JVM上提供了一个JavaScript实现。JavaScript是动态类型的面向对象语言（但和Java实现面向对象的方式截然不同）。Rhino既支持编译模式也支持解释模式，随Java 7一起发布（具体细节请参见com.sun.script.javascript包）。

最早的JVM语言

很难确定最早的JVM语言（除Java之外）是什么。可以肯定的是在1997年前后就出现了Kawa语言，它是一种Lisp语言。在那之后这些语言几乎呈现了爆炸式增长，因此追踪它们的历史太难了。

在编写本书时，猜测至少有200种JVM语言应该是合理的。不能说所有语言都很活跃或得到了广泛应用，但这个数字起码能表明JVM是一个非常活跃的语言开发和实现平台。

注意 在随Java 7推出的语言和VM规范里，所有对Java语言的直接引用都从VM规范中去掉了。Java现在只是运行在JVM上的众多语言中的普通一员，它不再享有特权了。

就像我们在第5章中讨论的，能让这么多不同的语言运行在JVM上的关键技术是类文件格式。任何能产生类文件的语言都可以认为是JVM上的编译型语言。

我们接下来会讨论多语言编程怎么变成了让Java程序员感兴趣的领域。我们会解释基本概念，为什么要给我们的项目选择一种备选的JVM语言以及如何操作。

7.3 JVM 上的多语言编程

“JVM上的多语言编程”这种说法还挺新颖的。这种说法是为了描述那些以Java代码为核心，但还用了一种或多种其他非Java JVM语言的项目。多语言编程通常是一种关注点分离的形式。如图7-3所示，非Java技术的作用可以分为三个层次。这张图有时被称为多语言编程金字塔，这要归功于Ola Bini。

金字塔中有三个明确的层次：特定领域层、动态层和稳定层。

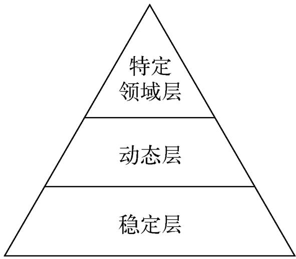


图7-3　多语言编程金字塔

多语言编程的秘密

多语言编程之所以有意义，是因为不同的代码片段有不同的生存期。银行里的风险引擎可能会持续运行五年以上；而网站上的JSP页面可能只有几个月；最短命的启动代码可能只"活"几天。代码"活"得时间越长，越靠近金字塔的底部。

它代表了不同侧重点的相互折中，比如底部更关注性能和全面测试，而顶部侧重的是灵活性和快速部署能力。

表7-1给出了这三个层次的更多细节。

表7-1　三层多语言编程金字塔

名　　称	描　　述	例　　子
特定领域层	特定领域语言。与应用程序领域的特定部分结合非常紧密	Apache Camel DSL、Drools、Web模板
动态层	开发速度快、生产率高、功能灵活部署	Groovy、Jython、Clojure
稳定层	核心功能、稳定、经过良好测试、性能高	Java、Scala

这些层次中有特定的模式，静态类型语言更倾向于稳定层的任务。相反，能力不是那么强、通用性比较低的技术在金字塔的顶部更容易找到自己的位置。

金字塔中部给动态语言留出了很多位置。这也是最灵活的一层，大多数情况下在动态层内部或者在动态层和相邻层之间有重叠。

我们要对这张图继续深挖，看看Java语言为什么不是金字塔所有层次的最佳选择。接下来我们先来看看为什么要考虑非Java语言，然后给出一些选择非Java语言的重要标准。

7.3.1　为什么要用非Java语言

Java作为一种通用、静态类型的编译型语言有很多优势。这些品质使它成为实现稳定层功能的绝佳选择。但同样的特性放到金字塔上层就会变成负担。比如说：

- ❑ 重新编译太费工了；
- ❑ 静态类型不够灵活，重构起来时间可能会比较长；

- 部署的动静太大；
- Java的语法天然不适用于生产DSL。

Java项目重新编译的时长迅速攀升到了90秒到2分钟。这个长度足以严重打断开发人员的思路，并且对于只在生产环境中存活几个星期的代码来说，这种开发方式太糟糕了。

比较务实的办法是利用Java丰富的API和类库完成稳定层的繁重工作。

注意 如果你刚开始一个新项目，可能会发现其他稳定层语言（比如Scala）也具备非常重要的特性（比如卓越的并发支持）。然而在大多数情况下，不应该用其他种类的稳定语言重写正在使用的稳定层代码。

这时你可能会纳闷："每一层都会面临什么样的编程挑战，我该选哪种语言？"一个优秀的Java开发人员知道，根本没有所谓的银弹，但当我们面对选择时，的确有一定的评估标准。我们不可能在这里把每个可能的选项都讨论到，所以在剩下的章节中，我们会集中讨论三种大多数Java开发人员可能都会面临的选择。

7.3.2 崭露头角的语言新星

接下来我们会挑三种，在我们看来可能最有生命力和影响力的语言。这些JVM语言（Groovy、Scala和Clojure）在多语言程序员心目中已经有了相当的分量。这三种语言为什么会得到大家的青睐？且听我们一一道来。

7

1. Groovy

Groovy语言是James Strachan在2003年发明的。它是动态的编译语言，语法跟Java很像，但更灵活。它被广泛用做脚本语言和快速原型语言，并且经常是开发人员或团队首选的非Java语言调研对象。你可以把Groovy看做是动态层的语言，它以擅长构建DSL著称。第8章主要介绍Groovy。

2. Scala

Scala是一门面向对象的语言，但也支持函数式编程。它的起源可以追溯到2003年，当时Martin Odersky正在用Java做一个与泛型相关的项目，结果却催生了Scala。它和Java一样，是静态类型的编译语言，但和Java不同，它做了大量的类型推断工作。也就是说它经常给人以动态语言的感觉。

Scala从Java中借鉴了很多东西，并且它的设计"修正"了Java中几个长期以来困扰开发人员的问题。Scala可以做稳定层语言，并且有些开发人员认为它可能会取代Java成为"JVM上的下一个大语言"。第9章介绍Scala。

3. Clojure

Clojure是由Rich Hickey设计的，属于Lisp家族的语言。它从Lisp中继承了很多语法特性（包括大量的括号[①]）。就像大多数Lisp语言一样，它是动态类型的函数式语言。它是编译型语言，但

---

① Lisp的表达式是一个原子（atom）或表（list）：原子（atom）又包含符号（symbol）与数值（number）；表是由零个或多个表达式组成的序列，表达式之间用空格分隔，放入一对括号中。此处的括号应该是指内置的表达式。
——译者注

通常以源码形态发布（稍后解释）。Clojure还向它的Lisp核心中添加了相当可观的新特性（特别是在并发方面）。

Lisp通常被当做专家语言。Clojure在某种程度上来说要比其他Lisp语言容易掌握，然而这并不会影响其强大的力量（也非常适合测试驱动的开发风格）。但它可能还是徘徊在主流之外，只是狂热的爱好者手中的秘密武器，抑或遇到适合它的特殊工作才发光（比如有些金融应用程序发现它的功能组合非常有吸引力）。

Clojure通常被认为是动态层的语言，但由于它的并发支持以及其他一些特性，也能胜任很多稳定层语言的工作。第10章会重点介绍Clojure。

现在我们已经把可选择的一部分语言罗列出来了，接下来该讨论一下决定你做出选择的那些因素了。

7.4　如何挑选称心的非 Java 语言

一旦决定在项目中实验非Java语言，就要先把项目中的各个工作域分清楚：哪些属于稳定层、哪些属于动态层或特定领域层。表7-2中给出了分属各层的工作。

表7-2　适合稳定层、动态层或特定领域层的项目域

名　　称	说　　明
特定领域层	构建、持续集成、持续部署
	开发操作
	企业集成模式建模
	业务规则建模
动态层	快速Web开发
	原型
	交互式管理与用户控制台
	脚本
	测试（比如用于测试驱动或行为驱动的开发）
稳定层	并发代码
	应用容器
	核心业务功能

如你所见，这些备选语言的使用范围非常广泛。但确定用备选语言解决哪项工作只是开始，接下来还要评估用备选语言是否合适。下面是帮我们选择技术的一些标准。

- 是否为项目里的低风险区。
- 备选语言跟Java的交互操作是否容易。
- 备选语言是否有工具支持（如IDE支持）。
- 语言学习难度。
- 招聘这门语言的开发人员的难度。

我们会逐一深入探讨这些标准。对于自己应该回答的问题，你得做到心中有数。

7.4.1 低风险

假设你有一个核心的支付处理规则引擎，每天要处理一百万笔交易。这是一个大概运行了七年、稳定的Java软件，但并没做过太多测试，代码里有大量死角。对于引入新语言来说，支付处理引擎显然是高危区，更不用说它本来跑得好好的，而且测试没做到全面覆盖，开发人员也还没完全弄明白它是怎么回事儿。

但系统不可能只有核心部分。比如更完善的测试明显会对系统有帮助。Scala有一个非常好的测试框架：ScalaTest（我们会在第11章介绍），可以用来测试Java或Scala代码。开发人员能用它写出跟JUnit相似但简洁得多的测试代码。所以一旦度过了ScalaTest的学习曲线，开发人员就能非常高效地增加测试覆盖面。而且ScalaTest对于逐步在代码库中引入行为驱动开发这样的概念很有办法。在将来要对核心的某些部分进行重构或替换时，不管最终新的处理引擎是用Java还是用Scala写，能用上现代测试特性真的很有帮助。

或者假设你需要建一个Web控制台，以便操作员能管理支付处理系统后台一些不太重要的静态数据。开发人员都知道Struts和JSF，可对这两种技术都提不起兴趣。这是另外一个试用新语言和技术栈的低风险区。Grails是个很抢眼的选择（基于Groovy的Web框架，受Ruby on Rails启发）。开发人员在经过一些研究后（Matt Raible也做过一个非常有趣的调研），一致认为Grails是生产率最高的Web框架。

因为是集中在低风险区的有限试点上做实验，如果所尝试的技术栈不适合自己的团队或系统，经理可以随时终止项目，不用中断太久就可以转移到不同的交付技术上。

7.4.2 与Java的交互操作

你肯定不想把原来写的那些Java代码弃之不用！很多组织都是因为这个原因迟迟不肯引入新的编程语言。但因为备选语言是跑在JVM上的，所以可以充分发挥原有代码的作用，这样问题变成了怎么让已有代码库的价值最大化，而不是抛弃正在使用的代码。

JVM上的备选语言跟Java之间的互操作简单利落，当然也能部署到原先的环境中。在讨论这个问题时一定要请管理生产环境的同仁到场参与。在把非JavaJVM语言加入到系统中时，你需要充分运用他们的专业经验。这也有助于消除他们对支持新方案的担忧，还能降低风险。

注意 DSL一般都是用动态层语言构建的（某些情况下也有稳定层语言），所以它们大多数是通过其内置语言运行在JVM上。

有些语言跟Java交互操作起来更容易。我们发现最流行的JVM备选语言（比如Groovy、Scala、Clojure、Jython和JRuby）都跟Java互操作得很好（而且其中某些语言的集成做得非常棒，几乎天衣无缝）。如果你确实是个谨小慎微的人，可以先做几个实验，很快，也很容易，而且你肯定能明白集成是如何工作的。

我们以Groovy为例。在Groovy代码里可以用我们熟悉的`import`语句直接导入Java包。用基于Groovy的Grails框架可以快速搭建一个网站，而引用对象仍然是Java模型对象。反过来说，在Java代码里调用Groovy代码也非常容易，有各种各样的办法，得到的也是熟悉的Java对象。比如从Java里调用Groovy代码处理JSON数据，并让它返回一个Java对象。

7.4.3　良好的工具和测试支持

大多数开发人员一旦习惯了已有环境，就会低估他们能节省的时间。有强大的IDE和构建工具、测试工具帮他们快速生产出高质量的软件。Java开发人员多年来受益于优秀的支持工具，所以一定要记得其他语言的成熟度可能还没达到相同的水平。

有些语言（比如Groovy）在编译、测试和最终结果部署方面长期以来都有IDE的支持。而其他语言的工具可能羽翼未丰。比如说Scala的IDE就不像Java的那么好用，但Scala粉觉得它的强大和简洁完全可以弥补当前IDE的不足。

还有个相关的问题，当备选语言社区开发出供自己使用的强大工具（比如Clojure的构建工具Leiningen）后，可能不太好调整它去处理其他语言。这就是说开发团队要认真考虑该如何分配项目，特别是在部署各自独立但又相互关联的组件时。

7.4.4　备选语言学习难度

学一门新语言总归需要时间，而且如果开发团队不熟悉该语言的范式，时间会更长。如果新语言是面向对象的，并有类C的语法（比如Groovy），那大多数Java开发团队都能轻松掌握。

可如果偏离了这一范式，偏离程度越大，Java开发人员觉得越难学。Scala试图在面向对象和函数式两个世界之间架起一座桥梁，但这种融合对于大规模软件项目是否可行仍然没有定论。在特别流行的几种备选语言中，Clojure可能会带来不可思议的好处，但开发团队在学习Clojure的函数式属性和Lisp语法时，也需要非常多的再培训工作。

还有一种选择是看看重新实现已有语言的JVM语言。Ruby和Python都是非常成熟的语言，有大量的材料可供开发人员学习。这些语言的JVM替身对于想采用易学的非Java语言的开发团队来说，是不错的起点。

7.4.5　使用备选语言的开发者

组织必须考虑现实情况：他们不可能总能雇到前2%的人（不管他们在广告里怎么忽悠），而且开发团队的成员也不会整年一成不变。某些语言，比如Groovy和Scala，已经足够成熟了，所以有相当的开发人员可以招募。但像Clojure这样还在想办法推广自己的语言，要找到优秀的Clojure开发人员仍然不太容易。

警告　对重新实现的语言的警告：比如说，很多现有的用Ruby写的包和应用程序，只在原始的基于C的实现下测试过。这就是说要在JVM上使用它们可能会有问题。在选择平台时，如果你计划采用整个用重新实现的语言编写的技术栈，那就应该把额外的测试时间考虑在内。

重新实现的语言（JRuby、Jython等）在这个问题上很可能再一次发挥作用。简历上有JRuby的开发人员可能很少，但因为它就是JVM上的Ruby，而Ruby开发人员非常多（熟悉C版本的Ruby开发人员很容易掌握在JVM上运行导致的差异）。

要理解备选JVM语言的设计选择及限制，你要先理解JVM如何支持多种语言。

7.5 JVM 对备选语言的支持

一种语言要在JVM上运行可能有两种方式：

- 有一个产生类文件的编译器；
- 有一个用JVM字节码实现的解释器。

无论哪种方式，通常都会有个运行时环境为执行该语言编写的程序提供支持。图7-4展示了Java和一种典型非Java语言的运行时环境栈。

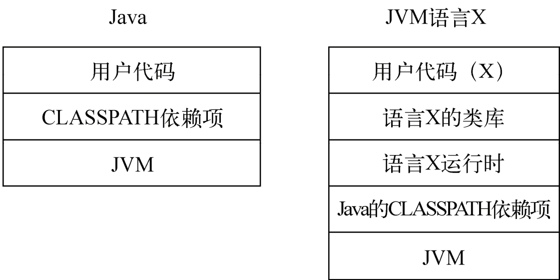


图7-4 非Java语言运行时支持

这些运行时支持系统复杂度各不相同，主要取决于给定的非Java语言运行时所要掌握的资源数量。几乎在所有情况下，运行时都会作为可执行程序类路径（classpath）上的一组JAR文件，并且要在程序执行开始之前启动。

本书的重点是编译型语言。至于Rhino等解释型语言，只是为了内容的完整性才提一下，所以我们不会在上面花太多篇幅。在本节剩下的内容中，我们会介绍备选语言所需的运行时支持（甚至还包括编译型语言），然后探讨编译器小说（compiler fiction）——那些可能不会出现在底层字节码中、由编译器合成的该语言特有的功能。

7.5.1 非Java语言的运行时环境

有一种评估语言运行时环境复杂度的简单办法：看运行时实现中JAR文件的大小。按这个标准，Clojure的运行时环境量级很轻，而JRuby语言则需要更多支持。

这个标准其实不是特别公平，因为有些语言把很多类库和功能都打包在标准发布版里了，而有些却不这么做。但如果不细究的话，它是个挺有用的经验。

通常来说，运行时环境是要帮助非Java语言的类型系统和其他特性符合期望的语义。备选语言对基本编程概念的看法有时和Java并不完全一致。

比如，Java的面向对象方式并不是放之四海而皆准的。在Ruby中，运行时可以往一个单独的对象里添加一个额外的方法，而这个方法在定义类时还不知道是什么，也不是在相同类的其他实例上定义的。JRuby的实现需要把这个属性（不太明白为什么叫“开放类”）照搬过来。而这种高级支持只能放在JRuby的运行时中。

7.5.2　编译器小说

语言的某些特性是由编程环境和高层语言合成的，在底层JVM中根本不存在。这些特性称为编译器小说。我们在第6章已经遇到过一个例子了——Java的字符串合并。

提示　你应该了解一下这些特性是如何实现的，否则你的代码可能跑得慢，甚至可能毁掉整个过程。有时运行时环境要做大量的工作来合成某个特性。

Java中的编译器小说还包括检查型异常和内部类（如有必要，内部类总会被转换成带有特殊合成访问方法的顶层类，如图7-5所示）。如果你曾经探究过JAR文件的内部（用`jar tvf`），见到过许多名字中有`$`的类，它们就是被取出并转换成“常规”类的内部类。

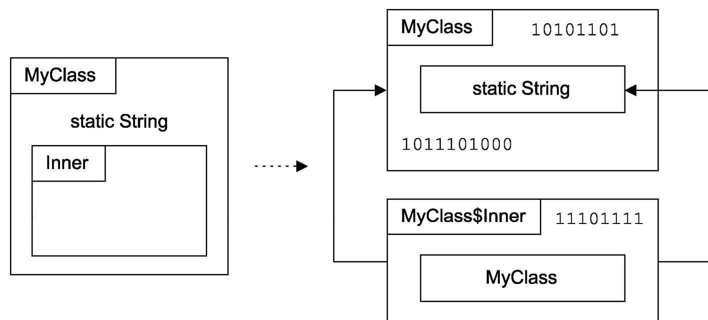


图7-5　用编译器小说实现的内部类

备选语言也有编译器小说。某些情况下，这些编译器小说甚至形成了语言的核心功能。我们来看两个重要的例子。

1. 函数是一等值

我们在7.1节介绍了函数式编程的关键概念——函数应该是能放到变量中的值。这通常说成“函数是一等值”。我们也指出Java对函数的建模方式并不太好。

本书第三部分讨论的所有非Java语言都把函数当做一等值。也就是说函数可以放在变量中、传给方法，并可以像操作其他任何值一样操作。JVM只能把类当做最小的代码和功能单元，所以现在所有的非Java语言都用小型匿名类作为函数的载体（不过这在Java 8中可能有所改变）。

解决源码和JVM字节码之间这种差异的办法是，记住对象只是把数据和操作数据的方法绑在一起的东西。请想象一个没有状态、只有一个方法的对象，比如第4章`Callable`接口的匿名实现

类。把这样一个对象放到变量中，作为参数传递，然后调用它的call()方法，这一切都很正常，像这样：

```
Callable<String> myFn = new Callable<String>() {
  @Override
  public String call() {
    return "The result";
  }
};

try {
  System.out.println(myFn.call());
} catch (Exception e) {
}
```

我们把异常处理忽略掉了，因为在这个例子中myFn的call()方法不可能抛出异常。

注意 在这个例子中，myFn变量是一个匿名类型，所以在编译后它看起来应该是个类似NameOf-EnclosingClass$1.class的东西。类的序号从1开始，并且编译器每遇到一个就加1。如果它们是动态创建的，并且数量很多（就像在JRuby语言中那样），会对存放类定义的PermGen内存区域造成压力。

尽管Java对此没有任何特殊的语法，但Java程序员经常用这个技巧创建匿名实现类。这就是说结果可能会有点冗长。我们所讨论的所有语言都提供了编写这些函数值（也叫函数字面值、匿名函数）的特殊语法。它们是函数式编程风格的支柱，Scala和Clojure做得都不错。

2. 多继承

还有一个例子，在Java（和JVM）中没办法表示实现的多继承。实现多继承的唯一办法就是使用接口，可它不允许有任何具体的方法。

相反，在Scala中特性机制（trait）允许把方法的实现混合到类中，所以它提供了不同的继承视图。我们会在第9章全面介绍。现在只要记住这种行为必须由Scala的编译器和运行时合成，在VM层面不提供这种特性。

对JVM上可用的不同类型的语言及其独特功能的实现方式就介绍到这里。

7.6 小结

JVM上的备选语言已经有了长足的发展。对于某些特定的问题，它们现在提供的解决方案要比Java好，并且还能与原来用Java技术实现的系统及投资兼容。这就是说，即便对于Java的推销者而言，Java也不总是所有编程任务的首选。

了解语言的不同分类方式（静态类型与动态类型、命令式与函数式、编译型与解释型）是为不同任务挑选正确语言的基础。

对于多语言程序员来说，编程语言大致可以分为三层：稳定层、动态层和特定领域层。Java和Scala这样的语言最好用来做稳定层的软件开发，而诸如Groovy和Clojure等其他语言更适合完成动态层或特定领域层的任务。

某些编程难题属于特定的层次，比如快速Web开发属于动态层，而建模企业消息属于特定领域层。

有必要再次强调一下，不要在已有生产系统的核心业务功能中引入新语言。对于核心功能区而言，支持级别高、测试覆盖率优异，并且有稳定的良好记录非常重要。与其从这里入手，还不如选一个风险低的领域部署备选语言。

不要忘了每个团队和项目都有自己独特的个性，这会影响选择语言时的决策。所以这个问题没有标准答案。在选择一门新语言时，项目经理和技术负责人必须把项目和团队的特性考虑在内。

一个都是经验丰富的技术人员组成的小团队可能会选择Clojure，因为它设计清晰、精巧并且强大（他们才不管概念的复杂性和招人的难度呢）。而一个Web团队，希望团队能快速扩充，能吸引年轻人，他们可能会因为生产率和储备相对较丰富的人才库而选择Groovy和Grails。

Groovy、Scala和Clojure是JVM语言中的领头羊。读完本书后，你能学到这三种最有前途的JVM备选语言的基础知识，并让自己的编程工具箱越来越有意思。

下一章我们会学习第一种语言：Groovy。

第 8 章 Groovy：Java的动态伴侣

本章内容

- 为什么学习Groovy
- Groovy的基本语法
- Groovy和Java之间的差别
- Groovy之于Java独有的特性
- Groovy为何又是脚本语言
- Groovy与Java的互操作性

Groovy是一种面向对象的动态类型语言，跟Java一样运行在JVM上。实际上，你可以把它看成是给Java静态世界补充动态能力的语言。Groovy项目最初是由James Strachan和Bob McWhirter在2003年末创建的，2004年其领导者变成了Guillaume Laforge。如今http://groovy.codehaus.org/上的Groovy社区仍在蓬勃发展。人们认为Groovy是继Java之后最流行的JVM语言。

受到Smalltalk、Ruby和Python的启发，Groovy已经实现了几个Java不具备的语言特性，比如：

- 函数字面值；
- 对集合的一等[①]支持；
- 对正则表达式的一等支持；
- 对XML处理的一等支持。

注意 在Groovy中，函数字面值也叫闭包。正如第7章所说的，它们是能够放进变量中的函数，能传递给方法，能像其他值一样操作。

那么我们为什么要用Groovy呢？如果你还记得第7章的多语言编程金字塔，也应该还记得Java不是解决动态层问题的理想语言。这些问题包括快速Web开发、原型设计、脚本处理以及其他很多问题。Groovy就是要解决这些问题。

这里有一个体现Groovy价值的例子。比如老板让你写一个Java例程，把一堆Java bean转成XML。用Java当然可以完成这个任务，而且有很多API和类库可以选用：

① 这里所说的“一等”是指内置到语言的语法中，不需要调用类库。

- ❑ Java 6中的Java Architecture for XML Binding（JAXB）和Java API for XML Processing（JAXP）；
- ❑ Codehaus上的XStream类库；
- ❑ Apache的XMLBeans类库。

当然还不止这些……

这个过程很费工。比如说，要在JAXB下输出Person对象对应的XML，你必须写出：

```
JAXBContext jc = JAXBContext.newInstance("net.teamsparq.domain");
ObjectFactory objFactory = new ObjectFactory();
Person person = (Person)objFactory.createPerson();
person.setId(2);
person.setName("Gweneth");
Marshaller m = jc.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
m.marshal(person, System.out);
```

Person类必须是标准的Java bean，有完整的获取和设置方法。

Groovy的方式与此不同，因为它把XML当做一等公民对待。这是上面那段代码的Groovy实现：

```
def writer = new StringWriter();
def xml = new groovy.xml.MarkupBuilder(writer);
xml.person(id:2) {
  name 'Gweneth'
  age 1
}
println writer.toString();
```

看到了吧，用这种语言写代码非常快，并且它和Java很像，Java开发人员很容易掌握。

Groovy还可以帮你减少套路代码的编写工作，比如Groovy处理XML和循环遍历集合的方式要比Java更简洁。因为Groovy跟Java的互操作性很好，所以在Java中很容易利用Groovy的动态性和语言特性。

因为跟Java的语法很像，所以对于Java开发人员来说，Groovy的学习曲线很平滑，并且只要有Groovy的JAR就可以开始了。希望你看完本章后能跟新语言拍档默契！

既然Groovy在由JVM执行之前经过了完整的分析、编译和生成过程，有些开发人员会想：“为什么它不能在编译时把那些明显的错误挑出来呢？”要记住Groovy是动态语言，它的类型检查和绑定都是在运行时做的。

Groovy的性能

如果你的软件性能要求很严格，Groovy语言并不是最好的选择。Groovy的对象都扩展自GroovyObject，它有一个invokeMethod (String name, Object args)方法。Groovy的方法不能像Java中那样直接调用，而是通过之前提到的invokeMethod (String name, Object args)方法执行。这个方法会自行执行一些反射调用和查找，这自然会降低处理速度。Groovy语言的开发者已经做了一些优化，而且在接下来的新版本中，Groovy会利用JVM中新的字节码invokedynamic做更多优化。

Groovy在做某些重活时还是靠Java，并且它要调用已有的Java类库很容易。因为Groovy能和Java一起使用它的动态类型和新语言特性，所以它是一种卓越的快速原型设计语言。Groovy还能作为脚本处理语言使用，因此它以Java的灵活、动态伴侣而著称。

本章一开始会跑几个简单的Groovy例子。一旦你熟悉了简单Groovy程序的运行，就可以开始学习Groovy的具体语法了，还有对于Java开发人员来说比较难缠的Groovy内容。本章接下来就深挖一下Groovy的真金真银，讨论Java不具备的几个语言特性。最后，你可以学习在JVM上混合使用Java和Groovy代码，成为一名多语言程序员。

8.1 Groovy 入门

如果你还没装Groovy，请先参照附录C在你的机器中把它搭起来，然后再编译和运行本章的第一个例子。

本节会向你展示如何用命令行编译和执行Groovy，以便你在任何操作系统上都能应用自如。我们还会介绍Groovy控制台，一个宝贵的、操作系统无关的暂存器环境，非常适合用来练手。

装好了吗？那我们就来编译一些Groovy代码，让它们跑起来吧！

8.1.1 编译和运行

这里有些你应该了解的Groovy命令行工具，特别是编译器（groovyc）和运行时执行器（groovy）。它们两个基本上就相当于javac和java。

8

为什么代码示例的编码风格变了？

越往后，本章中的示例代码的语法和语义越像纯粹地道的Groovy。希望这样能让你更容易从Java向Groovy转移。再向你推荐一本非常优秀的书：Kenneth A. Kousen编著的*Making Java Groovy*（Manning，2012）。

我们来看一个简单的Groovy脚本，它可以输出下面的内容[①]，也借此熟悉一下命令行工具：

```
It's Groovy baby, yeah!
```

打开命令行提示符，执行如下操作。

(1) 随便找个目录，在里面创建一个HelloGroovy.groovy文件。

(2) 编辑这个文件，加上这一行：

```
System.out.println("It's Groovy baby, yeah!");
```

(3) 保存HelloGroovy.groovy。

(4) 用下面这个命令编译它：

```
groovyc HelloGroovy.groovy
```

① 感谢《王牌大贱谍》!

(5) 用下面这个命令运行它：

```
groovy HelloGroovy
```

提示　如果Groovy源文件在CLASSPATH下，可以跳过编译。如果需要，Groovy运行时会先在源文件上执行groovyc。

恭喜，你刚刚运行了有生以来第一行Groovy代码！

跟Java一样，你可以在命令行中编写、编译和执行Groovy代码，但要处理CLASSPATH之类的事情时，你很快就会觉得这么做太笨了。主流的Java IDE（Eclipse、IntelliJ和NetBeans）对Groovy的支持都很好，但Groovy也提供了一个控制台供你运行代码。这个控制台非常适合快速演练小型解决方案或原型，因为用它比用正式的IDE快得多。

8.1.2　Groovy控制台

本章会用Groovy控制台运行示例代码，因为它是一个好用、轻量的IDE。要启动控制台，请在命令行中执行groovyConsole。

它会弹出一个类似图8-1这样的独立窗口。

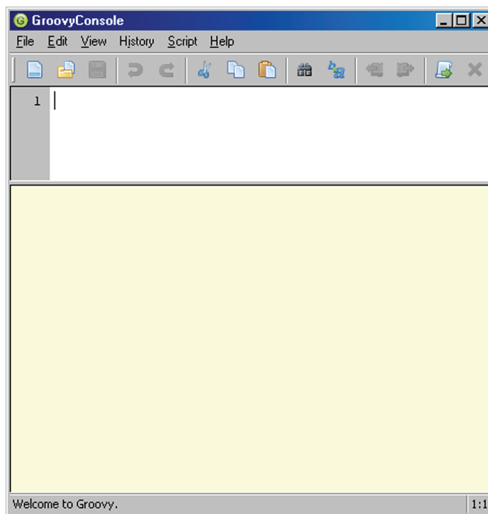


图8-1　Groovy控制台

首先，你应该取消勾选View（视图）菜单中的Show Script in Output（在输出中显示脚本）选项。这会让输出简单一点儿。现在你可以运行一下前面那个例子中的Groovy代码，以确保控制台能正常工作。在控制台的顶部面板中输入下面这行代码：

```
System.out.println("It's Groovy baby, yeah!");
```

然后点击Execute Script（执行脚本）按钮，或者用快捷键Ctrl-R。Groovy控制台就会在底部面板中显示如下输出：

```
It's Groovy baby, yeah!
```

如你所见，输出面板显示了刚刚执行的那个表达式的计算结果。

现在你已经知道如何快速执行Groovy代码了，是时候学一些Groovy的语法和语义了。

8.2 Groovy 101：语法和语义

上一节只写了一行Groovy语句，没有任何类或方法之类的结构（用Java时会需要）。实际上你写的是一个Groovy脚本。

Groovy脚本

跟Java不同，Groovy的源码可以当做脚本执行。比如说，如果你有一段代码放在类定义之外，那段代码还是可以执行。像其他动态脚本语言（比如Ruby或Python）一样，Groovy脚本在JVM上执行之前要在内存中经过完整的分析、编译和生成过程。任何能在Groovy控制台执行的代码都可以保存到.groovy文件，经过编译后，就可以作为脚本运行。一些开发人员已经用Groovy脚本取代了shell脚本，因为它们功能更强，更易于编写，并且只要装了JVM，就可以在任何平台上运行。给你一个性能方面的小提示，请使用groovyserv类库，它会启动JVM和Groovy扩展，让脚本运行得更快。

Groovy的一个关键特性是可以使用跟Java中一样的结构，语法也类似。为了突出这种相似性，请在Groovy控制台中执行下面这段类似Java的代码：

```
public class PrintStatement
{
  public static void main(String[] args)
  {
    System.out.println("It's Groovy baby, yeah!");
  }
}
```

结果和前面那个只有一行的Groovy脚本一样，都是输出"It's Groovy baby, yeah!"。除了使用Groovy控制台，你还可以把源码放到PrintStatement.groovy源文件中，用groovyc编译它，然后用groovy执行。换句话说，你能像Java中那样带着类和方法编写Groovy源码。

提示 在Groovy中几乎可以使用所有Java普通语法，所以while/for循环、if/else结构、switch语句等，都会按你期望的方式工作。所有新语法及主要差异都会在本节及相应章节中重点阐述。

随着本章内容的深入，我们会向你介绍Groovy特有的语法惯用语，例子也会从类似Java的语法向更纯粹的Groovy语法转变。你已经习惯了结构沉重的Java代码，再见到像脚本一样简洁的Groovy语法，很容易发现两者的差异。

本节的剩余部分会介绍Groovy的基本语法和语义，以及它们为什么能帮助开发人员。具体来说，我们会探讨：

- 默认导入；
- 数字处理；
- 变量、动态与静态类型，以及作用域；
- 列表与映射的语法。

首先，理解Groovy提供了哪些开箱即用的东西很重要。我们先来看看Groovy脚本或程序的默认导入。

8.2.1　默认导入

Groovy会默认导入一些语言包和工具包，以提供基本的语言支持。Groovy还会导入一系列的Java包，以便为其初始功能提供更广泛的基础。下面这个导入列表总是隐含在Groovy代码之中：

- `groovy.lang.*`
- `groovy.util.*`
- `java.lang.*`
- `java.io.*`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.net.*`
- `java.util.*`

要使用更多的包和类，可以像Java一样用`import`语句。比如要从Java中得到所有`Math`类，只要在Groovy源码里加上`import java.math.*;`就行了。

设置可选的JAR文件

为了添加功能（比如内存数据库及其驱动），可以在Groovy安装中添加可选JAR。Groovy为此提供了一个惯用语：通常是在脚本中使用`@Grab`注解。另外一种办法（在你仍在学习Groovy时）是效仿Java，把JAR文件加到`CLASSPATH`中。

下面就来使用一下默认的语言支持，并看看Java和Groovy在数字处理上的差异。

8.2.2　数字处理

Groovy能动态计算数学表达式，并且它采用最小意外原则。这一原则在处理浮点数时（比如

3.2）尤其明显。Groovy在底层用Java中的`BigDecimal`表示浮点数，但它会确保`BigDecimal`的行为尽量符合开发人员的期望。

1. Java和BigDecimal

我们来看一个经常会让开发人员头疼的数字处理问题。在Java中，如果在`BigDecimal 3`上加0.2，你觉得答案应该是什么？缺乏经验的Java开发人员在没看Javadoc的情况下很可能会执行下面这种代码，它会返回一个极其恐怖的结果：3.2000000000000001110223024625156540423631 6680908203125。

```
BigDecimal x = new BigDecimal(3);
BigDecimal y = new BigDecimal(0.2);
System.out.println(x.add(y));
```

经验丰富的Java开发人员知道最好用`BigDecimal(String val)`，而不是用将数字作为参数的`BigDecimal`构造方法。以字符串为参数的构造方法写出来的代码会产生预期答案3.2：

```
BigDecimal x = new BigDecimal("3");
BigDecimal y = new BigDecimal("0.2");
System.out.println(x.add(y));
```

这有点悖于常理，所以Groovy默认采用了以字符串为参数的构造方法，解决了这一问题。

2. Groovy和BigDecimal

在Groovy中处理浮点数（在底层用`BigDecimal`表示）时，会自动使用以字符串为参数的构造方法，3 + 0.2会得到3.2。你可以在Groovy控制台中输入下面的指令亲自证实一下：

```
3 + 0.2;
```

你会发现Groovy对BEDMAS[①]的支持是正确的。并且在需要时能无缝切换数字类型（比如`int`和`double`）。

用Groovy进行数学运算比Java简单。如果你想了解底层细节，可以访问http://groovy.codehaus.org/Groovy+Math，那里有所有的细节信息。

接下来我们学习Groovy如何处理变量和作用域。因为Groovy的动态性和执行脚本的能力，它在这方面的语义规则和Java稍有不同。

8.2.3 变量、动态与静态类型、作用域

因为Groovy是一种能作为脚本语言的动态语言，所以你要清楚动态类型和静态类型一些细微差别，还需要了解Groovy如何限定变量的作用域。

提示 如果你意在让Groovy代码与Java互操作，它也能在可能的情况下使用静态类型，因为它简化了类型重载和调度机制。

---

① 回想起你在学校的日子了吧！BEDMAS表示括号、次方、除法、乘法、加法和减法，是我们计算数学题目时所要遵循的顺序（先计算括号和次方，再计算乘除，最后计算加减）。由于地区不同，你的记忆中可能是BODMAS或PEMDAS。

首先你要理解Groovy动态类型和静态类型的差别。

1. 动态类型与静态类型

Groovy是动态语言，所以不必指定变量的类型，变量的类型是在声明（或返回）时确定的。比如说，你可以把一个Date赋值给变量x，然后紧接着再用不同的类型给x赋值。

```
x = new Date();
x = 1;
```

用动态类型能让代码更简洁（忽略显而易见的类型信息），反馈更快，并且很灵活，可以在一个变量上赋予不同类型的对象来完成工作。对于那些想对自己使用的类型更有把握的人，Groovy也确实支持静态类型。比如：

```
Date x = new Date();
```

如果声明了静态类型变量，在用不正确的类型值对它赋值时，Groovy能检查出来。比如：

```
Exception thrown

org.codehaus.groovy.runtime.typehandling.GroovyCastException: Cannot cast
     object 'Thu Oct 13 12:58:28 BST 2011' with class 'java.util.Date' to
     class 'double'
...
```

在Groovy控制台中运行下面的代码，就可以重现上面的输出。

```
double y = -3.1499392;
y = new Date();
```

如你所料，Data类型的值不能赋给double变量。Groovy中的动态和静态类型都讨论到了，那作用域呢？

2. Groovy中的作用域

对于Groovy里的类，其作用域跟Java一样，类、方法、循环作用域的变量，它们的作用域都跟你想的一样。但涉及Groovy脚本时，这个话题就变得比较有意思了。

提示　记住，作为脚本的Groovy代码不在平常的类和方法结构中。8.1.1节已经给过一个例子了。

简单说，Groovy脚本有两种作用域。

- 绑定域，绑定域是脚本的全局作用域。
- 本地域，本地域就是变量的作用域局限于声明它们的代码块。对于在脚本代码块内声明的变量（比如在脚本的顶部），如果是定义过的变量，其作用域就是定义它的本地域。

能在脚本中使用全局变量可以极大提高代码的灵活性。它和Java中类范围内的变量有点像。定义变量是指被声明为静态类型，或用特殊的def关键字定义的变量（表明它是未确定类型的定义变量）。

在脚本中声明的方法访问不了本地域。如果你调用一个试图引用本地域中的变量的方法，会提示类似下面的错误消息：

```
groovy.lang.MissingPropertyException: No such property: hello for class:
     listing_8_2
...
```

下面是产生该异常的代码，说明了作用域的这个问题。

```
String hello = "Hello!";
void checkHello()
{
  System.out.println(hello);
}
checkHello();
```

如果用`hello = "Hello!"`换掉上面代码里的第一行，这个方法可以成功输出“Hello”。因为`hello`不再定义为`String`，它现在的作用域是绑定域。

除了编写Groovy脚本时的这些差异，动态和静态类型、作用域、变量声明都跟你想的完全一样。接下来我们去看看Groovy内置的集合（列表和映射）支持。

8.2.4 列表和映射语法

Groovy把列表和映射（包括集合）结构当做语言中的一等公民对待，所以没必要像Java那样显式声明`List`和`Map`结构。也就是说，Groovy中的列表和映射在底层是由Java `ArrayList`和`LinkedHashMap`实现的。

使用Groovy语法最大的优势在于可以省掉很多套路化的代码，让代码更简洁，但丝毫不影响可读性。

Groovy用方括号`[]`指定和使用列表结构（是不是想起了Java中的原生数组语法）。下面的代码展示了如何引用第一个元素（`Java`），获取列表大小（4），以及将列表设置为空`[]`。

```
jvmLanguages = ["Java", "Groovy", "Scala", "Clojure"];
println(jvmLanguages[0]);
println(jvmLanguages.size());
jvmLanguages = [];
println(jvmLanguages);
```

看，Groovy将列表作为一等公民处理要比用`java.util.List`及其实现类的代码轻量得多。

因为Groovy是动态类型语言，我们可以把不同类型的值保存在列表（或映射）中，所以下面的代码也是正确的：

```
jvmLanguages = ["Java", 2, "Scala", new Date()];
```

Groovy处理映射也跟这差不多，用`[]`符号，并用冒号（`:`）来分开键/值对。以映射.键的方式引用映射中的值。下面的代码通过相应的操作展示了这些功能：

- 引用键`"Java"`的值`100`；
- 引用键`"Clojure"`的值`"N/A"`；
- 将键`"Clojure"`的值变成`75`；
- 将映射设为空（`[:]`）。

8

```
languageRatings = [Java:100, Groovy:99, Clojure:"N/A"];
println(languageRatings["Java"]);
println(languageRatings.Clojure);
languageRatings["Clojure"] = 75;
println(languageRatings["Clojure"]);
languageRatings = [:];
println languageRatings;
```

提示　你有没有注意到映射里的键是不带引号的字符串？为了让代码更简洁，Groovy对这个语法也做了调整，映射键的引号可用可不用。

这种写法很直观，用起来也舒服。Groovy把对映射和列表内置支持的概念更进了一步。

还有一些语法技巧，比如引用集合中一定范围内的元素，甚至可以用负索引引用最后一个元素。下面的代码引用了列表中的前三个元素（[Java, Groovy, Scala]）和最后一个元素（Clojure）。

```
jvmLanguages = ["Java", "Groovy", "Scala", "Clojure"];
println(jvmLanguages[0..2]);
println(jvmLanguages[-1]);
```

现在，我们已经了解了Groovy的一些基本语法和语义。但在真正使用Groovy之前，还需要学习更多内容。下一节会更深入地探讨Groovy的语法和语义，重点讲解Java开发人员学习Groovy过程中那些“难缠的内容”。

8.3　与 Java 的差异——新手陷阱

目前你基本上已经熟练掌握Groovy的基本语法了，当然其中部分原因是它跟Java语法很像。但这种相似有时可能是“诈”，所以本节来讨论那些经常困扰Java开发人员的语法。

Groovy有大量可以省略的语法，比如：

- 语句结束处的分号；
- 返回语句（return）；
- 方法参数两边的括号；
- public访问限定符。

这类设计是为了让源码更简洁，在快速设计原型时都能体现出优势来。

其他修改包括去掉已检查和未检查异常之间的区别，相等概念的替代办法，以及不再使用内部类。我们先从最简单的开始：可选的分号和返回语句。

8.3.1　可选的分号和返回语句

在Groovy中，语句结束处的分号（;）是可选的，除非一行中有多条语句，否则都可以省略。

此外，从方法中返回对象或值时不必使用return关键字。Groovy会自动返回最后一个表达式的计算结果。

代码清单8-1演示了这些可选语法，并返回了方法中最后一个表达式的计算结果3。

代码清单8-1　分号和返回语句可以省略

```
Scratchpad pad = new Scratchpad()
println(pad.doStuff())

public class Scratchpad
{
  public Integer doStuff()
  {
    def x = 1
    def y; def String z = "Hello";
    x = 3
  }
}
```

没有分号

没有返回语句

上面的代码看起来跟Java还是很像，而Groovy的简洁风格其实还要更强。接下来你会看到Groovy如何省略方法参数两边的括号。

8.3.2　可选的参数括号

如果Groovy里的方法调用至少有一个参数，并且没有二义性，则可以省略括号。也就是说下面的代码

```
println("It's Groovy baby, yeah!")
```

可以写成

```
println "It's Groovy baby, yeah!"
```

代码变得更简洁了，可读性仍然没受影响。

下一个特性是可选的`public`访问限定符，再用上它，Groovy代码看起来就不太像Java了。

8.3.3　访问限定符

优秀的Java开发人员都知道确定类、方法和变量的访问级别是面向对象设计的重要组成部分。跟Java一样，Groovy也有`public`、`private`和`protected`级别；但和Java不同，Groovy的默认访问级别是`public`。所以我们把代码清单8-1改一下，去掉一些默认的`public`限定符，加几个`privte`限定符，如代码清单8-2所示。

代码清单8-2　`public`是默认访问限定符

```
Scratchpad2 pad = new Scratchpad2()
println(pad.doStuff())

class Scratchpad2
{
  def private x;
  Integer doStuff()
  {
```

没有指定**`public`**访问限定符

```
      x = 1
      def y; def String z = "Hello";
      x = 3
    }
  }
```

继续语法精简的主题，作为一名Java开发人员，你对用在方法签名中抛出已检查异常的throws语句熟不熟悉？

8.3.4　异常处理

跟Java不同，Groovy不区分已检查异常和未检查异常。Groovy编译器会忽略方法签名中的所有throws语句。

在保证源码可读的前提下，Groovy采用了一些快捷语法来简化代码。接下来看一个有严重语义影响的语法变化：相等操作符。

8.3.5　Groovy中的相等

遵循最小意外原则，Groovy把==当做Java中的equals()方法。这是直觉式开发人员的又一项福利，他们不必再像用Java时为原始类型和对象倒腾==和equals()。

检查真实的对象是否相等，需要使用Groovy内置的is()函数。这一规则有个例外，就是你仍然可以用==来检查一个对象是否为null。代码清单8-3说明了这些特性。

代码清单8-3　Groovy中的相等

```
Integer x = new Integer(2)
Integer y = new Integer(2)
Integer z = null
if (x == y)
{
  println "x == y"                 // 隐含的equals()调用
}

if (!x.is(y))
{
  println "x is not y"             // 检查对象是否相等
}

if (z.is(null))
{
  println "z is null"              // 用is()检查是否为null
}

if (z == null)
{
  println "z is null"              // 检查是否为null
}
```

当然，如果你喜欢，仍然可以用equals()方法检查相等关系。

最后，还有一个应该简单提一下的Java结构——内部类，Groovy中它基本被一个新的语言结构取代了。

8.3.6 内部类

Groovy支持内部类，但大多数情况下我们应该用函数字面值替代它。下一节讨论函数字面值，它是一个很强的现代编程结构，应该占用更多篇幅介绍。

用Groovy可以写出更简洁的代码，而且并不影响代码的可读性，如果你喜欢，仍然可以继续使用Java的（大多数）语法结构。接下来，你会看到一些Java还不具备的Groovy语言特性。其中的某些特性很可能就是你选用Groovy的关键，比如XML处理。

8.4 Java 不具备的 Groovy 特性

Groovy具备一些Java没有的语言特性，起码Java 7还没有。优秀的Java开发人员就是在这些问题上需要向新语言求助，希望能以更优雅的方式解决它们。本节就探索几个这样的特性，包括：

- GroovyBean，更简单的bean；
- 用操作符`?.`实现`null`对象的安全访问；
- 猫王[①]操作符（Elvis operator），更短的`if/else`结构；
- Groovy字符串，更强的字符串抽象；
- 函数字面值（即闭包），把函数当做值传递；
- 对正则表达式的本地支持；
- 更简单的XML处理。

我们会从GroovyBean开始，因为Groovy代码中经常见到它们。作为一名Java开发人员，你可能有点儿疑心，因为按JavaBean的标准来衡量的话，它们不太完整。但请你放心，GroovyBean很完整，分毫不差，并且用起来更方便。

8.4.1 GroovyBean

GroovyBean很像JavaBean，不过省略了显式声明的获取和设置方法，提供了自动构造方法，并允许你用点号（.）引用成员变量。如果需要把某个获取方法或设置方法设为`private`，或者希望改变默认的行为，可以显式声明那个方法，并按你的想法修改它。自动构造方法只是一个用来构造GroovyBean、传入与GroovyBean的成员变量对应的参数的映射。

不论是不辞劳苦自己输入获取方法和设置方法，还是用IDE生成，所有这些都省去了我们处理JavaBean时所要编写的大量套路化代码。

我们以一个角色扮演游戏（RPG）[②]里的`Character`类为例来看一下GroovyBean是如何工作的。代码清单8-4会输出`STR[18], WIS[15]`，这是代表GroovyBean力量和智慧的成员变量。

---

① Elvis Aron Presley（1935—1977），美国摇滚乐史上影响力最大的歌手，有摇滚乐之王的誉称。——译者注

② 这里大力推荐一下PCGen（http://pcgen.sf.net），对于RPG粉来说真是个非常好的开源项目。

代码清单8-4　探索GroovyBean

```
class Character
{
  private int strength
  private int wisdom
}

def pc = new Character(strength: 10, wisdom: 15)
pc.strength = 18
println "STR [" + pc.strength + "] WIS [" + pc.wisdom + "]"
```

它的行为跟Java里的JavaBean非常相似（封装性得以保留），而语法更精简。

> **提示**　可以用`@Immutable`注解使GroovyBean不可变（意思是它的状态不可修改）。这对于传递线程安全的数据结构很有用，在并发代码中用起来更安全。第10章讨论闭包时我们还会进一步讨论不可变数据结构的概念。

接下来我们会转向Groovy检查`null`引用的能力。这会进一步减少套路化代码，以便你可以更快地把想法变成原型。

8.4.2　安全解引用操作符

`NullPointerException`[①]（NPE）是所有Java开发人员都挥之不去的梦魇（很不幸）。为了避开NPE，Java程序员通常都会在引用对象之前检查一下它是否为`null`，特别是在他们不能保证所处理的对象不是`null`的情况下。如果你准备在Groovy中延续那种开发风格，为了遍历一个`Person`对象列表，最终编写的代码可能像下面这样（只是输出“Gweneth”）。

```
List<Person> people = [null, new Person(name:"Gweneth")]
for (Person person: people) {
  if (person != null) {
    println person.getName()
  }
}
```

Groovy引入了安全解引用运算符，用`?.`符号帮你去掉一些套路化的“如果对象为`null`”检查代码。在使用这个符号时，Groovy引入了一个特殊的`null`结构，表示“什么也不做”，而不是真的引用`null`。

在Groovy中，可以用安全解引用语法重写上面的代码：

```
people = [null, new Person(name:"Gweneth")]
for (Person person: people) {
  println person?.name
}
```

Groovy函数也支持这种安全解引用，所以Groovy的默认集合方法（比如`max()`方法），能自动处理好`null`引用。

---

① Java最大的憾事就是没据实把这个叫做`NullReferenceException`，本书的一位作者对此一直颇多怨言！

接下来是猫王操作符，看起来和安全解引用差不多，但它是用来减少某些if/else结构中的代码的。

8.4.3 猫王操作符

用猫王操作符（?:）可以把带有默认值的if/else结构写得极其短小。为什么叫猫王？因为这个符号看起来明显很像猫王鼎盛时期梳的大背头①。用猫王操作符不用检查null，也不用重复变量。

假设你要检查王牌大贱谍是不是活跃的侦探。在Java中可能要用三元操作符：

```
String agentStatus = "Active";
String status = agentStatus != null ? agentStatus : "Inactive";
```

Groovy能缩短这个语句，是因为它能在需要时将类型强制转换为boolean，比如if语句的条件判断。在前面的代码中，Groovy把String转换为boolean，假如String是null，它会被转换成Boolean值false，所以可以省略null检查。因而前面的代码可以写成这样：

```
String agentStatus = "Active"
String status = agentStatus ? agentStatus : "Inactive"
```

但这样还是要重复agentStatus变量，Groovy可以让我们不再重复输入。用猫王操作符可以去掉重复的变量名：

```
String agentStatus = "Active"
String status = agentStatus ?: "Inactive"
```

第二个agentStatus没了，代码更简洁了。

好了，现在该去看看Groovy字符串了，看看它们跟Java常规String有什么不同。

8.4.4 增强型字符串

Groovy有一个String类的扩展类GString，它比Java中标准的String强，也更灵活。

尽管双引号也有效，但按照惯例，普通字符串是用开闭两个单引号定义的。比如：

```
String ordinaryString = 'ordinary string'
String ordinaryString2 = "ordinary string 2"
```

而GString必须用双引号定义。对于开发人员来说，使用它最大的好处是可以包含可在运行时计算的表达式（用${}）。如果GString随后被转为普通字符串（比如传给了println），GString中的表达式都会被替换为其计算结果。比如：

```
String name = 'Gweneth'
def dist = 3 * 2
String crawling = "${name} is crawling ${dist} feet!"
```

其中的表达式计算后被转到可以调用toString()的Object上，或是函数字面值上。（请参见http://groovy.codehaus.org/Strings+and+GString了解关于函数字面值和GString规则的细节。）

① 本书的作者都郑重声明，我们根本不知道猫王在鼎盛时期长什么样。我们真没那么老，不开玩笑！

警告　GString的底层并不是Java中的String！尤其不应该把GString作为映射中的键，或者比较它们是否相等。结果是不可预料的！

Groovy中另一个有点儿用的结构是三引号String或三引号GString，它们可以在源码中定义跨行字符串。

```
"""This GString
wraps over two lines!"""
```

接下来我们要向函数字面值进军了。由于最近几年业内兴起了对函数式语言的兴趣，这个编程技巧也成了一个热门话题。要弄懂函数字面值，可能需要动动脑筋。如果你没用过，也就是说如果这是你第一次用，也许你现在就该先起身将公爵杯加满自己喜欢的饮品。

8.4.5　函数字面值

函数字面值表示一个可以当做值传递的代码块，也可以像操作任何值一样操作。可以当参数传给方法，可以给变量赋值，等等。这个语言特性已经成为Java社区的讨论热点，但对于Groovy程序员来说，它们是标配的工具。

举例说明向来都是学习新概念的最好方法，我们先来看几个例子吧！

假设我们有一个普通的静态方法，要构建一个String来向作者或读者问好。我们用常规方式从这个类的外部调用该方法，如代码清单8-5所示：

代码清单8-5　一个简单的静态函数

```
class StringUtils
{
  static String sayHello(String name)        <-- 静态方法声明
  {
    if (name ==  "Martijn" || name == "Ben")
      "Hello author " + name + "!"
    else
      "Hello reader " + name + "!"
  }
}
println StringUtils.sayHello("Bob");         <-- 调用者
```

有了函数字面值，你不用方法或类结构也可以实现同样的功能，只要把代码放在函数字面值里。而函数字面值又可以赋值给一个变量，从而可以被传递和执行。

代码清单8-6把函数字面值赋值给sayHello，传入参数"Martijn"，并最终输出“Hello author Martijn!”。

代码清单8-6　使用简单的函数字面值

```
def sayHello =                               <-- 函数字面值赋值
{
  name ->                                    <-- ❶ 变量与处理逻辑分开
    if (name == "Martijn" || name == "Ben")
      "Hello author " + name + "!"
    else
```

```
        "Hello reader " + name + "!"
    }
    println(sayHello("Martijn"))
```

输出结果

注意函数字面值开始处的{。把传入函数字面值的参数跟处理逻辑分开的箭头（->）❶。最后是函数字面值结束处的}。

在代码清单8-6中，函数字面值的定义方式非常像方法的定义方式。因此你可能在想：“它们看起来也不是特别有用！”只有开始用它们创作（用函数方式思考）时，你才能真正发现它们的好，比如说跟Groovy对集合的内置支持结合起来之后，函数字面值会特别强大。

8.4.6 内置的集合操作

Groovy有几个可以用于集合（列表和映射）的内置方法。这种在语言层面对集合的支持，跟函数结合在一起，可以极大减少程序员在Java中必写的那些套路化代码；并且代码仍然很容易看懂，不影响维护。

表8-1是一些使用了函数字面值的内置函数。

表8-1 Groovy中的部分集合函数

方　法	描　述
each	遍历集合，对其中的每一项应用函数字面值
collect	收集在集合中每一项上应用函数字面值的返回结果（相当于其他语言map/reduce中的map函数）
inject	用函数字面值处理集合并构建返回值（相当于其他语言里map/reduce中的reduce函数）
findAll	找到集合中所有与函数字面值匹配的元素
max	返回集合中的最大值
min	返回集合中的最小值

Java编程过程中遍历集合，并对其中每个对象执行某种操作是很常见的任务。比如说，如果你想在Java 7中输出电影名称，很可能会写出如代码清单8-7所示的代码：[①]

代码清单8-7 在Java 7中输出一个集合

```
List<String> movieTitles = new ArrayList<>();
movieTitles.add("Seven");
movieTitles.add("Snow White");
movieTitles.add("Die Hard");

for (String movieTitle : movieTitles)
{
  System.out.println(movieTitle);
}
```

Java中有帮你少写代码的技巧，但不管怎样都要用某种循环结构手工遍历电影名称的List。

在Groovy里可以用内置的集合遍历功能（each函数），并且函数字面值可以减少大量你需要

① 不，我们可不会告诉你谁喜欢《白雪公主》（反正不是我俩）!

自己编写的代码。此外，这样还能反转列表和所要执行的算法之间的关系。不再是把集合传递到方法中，而是把方法传入到集合中！

下面的代码和代码清单8-7所做的工作完全一样，但只有短短的两行，很容易读懂：

```
movieTitles = ["Seven", "SnowWhite", "Die Hard"]
movieTitles.each({x -> println x})
```

实际上，如果使用隐含的it变量，这段代码还可以变得更精简，it变量可以用在单参的函数字面值中，代码如下所示[①]：

```
movieTitles = ["Seven", "SnowWhite", "Die Hard"]
movieTitles.each({println it})
```

看，这段代码简洁易读，并且效果和Java 7那个版本一样。

提示　只能介绍这么多了，如果你想研究更多例子，推荐你到Groovy的网站上去看看与集合相关的内容（http://groovy.codehaus.org/JN1015-Collections），或者读读Dierk König、Guillaume Laforge、Paul King、Jon Skeet和Hamlet D'Arcy合著的*Groovy in Action, second edition*（Manning, 2012），这是一本相当不错的书。

下一个语言特性是Groovy内置的正则表达式支持，你可能要花点儿时间才能熟悉，所以借着咖啡劲儿，我们赶紧来看看吧！

8.4.7　对正则表达式的内置支持

Groovy把正则表达式当做语言的一部分，所以用Groovy处理文本要比Java简单得多。表8-2中是Groovy可用的正则表达式语法，以及Java与之对应的东西。

表8-2　Groovy正则表达式语法

方　法	描述及Java中的对等物
~	创建一个模式（创建一个编译的Java Pattern对象）
=~	创建一个匹配器（创建一个Java Matcher对象）
==~	计算字符串（相当于在Pattern上调用Java match()方法）

假设你从一个硬件上收到了一些日志数据，要部分匹配其中一些错误日志。比如查找模式1010的实例，然后再找0101。在Java 7中，实现代码可能如下所示。

```
Pattern pattern = Pattern.compile("1010");
String input = "1010";
Matcher matcher = pattern.matcher(input);
if (input.matches("1010"))
{
  input = matcher.replaceFirst("0101");
```

① Groovy高手会说实际上还可以简化，一行足矣！

```
  System.out.println(input);
}
```

在Groovy中，每行代码都变短了，因为Pattern和Matcher对象是内置在语言中的。当然，输出（0101）还和原来一样，请看代码。

```
def pattern = /1010/
def input = "1010"
def matcher = input =~ pattern
if (input ==~ pattern)
{
  input = matcher.replaceFirst("0101")
  println input
}
```

Groovy支持完整的正则表达式语义，所采用的方式和Java一样，所以你熟悉的那种灵活性还在。

正则表达式跟函数字面值结合得也很好。比如分析String得到一个人的名字和年龄，并输出详细信息。

```
("Hazel 1" =~ /(\w+) (\d+)/).each {full, name, age
                              -> println "$name is $age years old."}
```

或许你应该借这个机会稍微放松一下，接下来我们马上就要探索一项完全不同的技术：XML处理。

8.4.8 简单的XML处理

Groovy有构建器的概念，用Groovy原生语法可以处理任何树型结构的数据。包括HTML、XML和JSON。Groovy理解开发人员想轻松处理这种数据的需求，所以提供了开箱即用的构建器。

XML：一种被滥用的语言

XML是一种卓越、详细的数据交换语言，但现在已经变得如洪水猛兽一般了。为什么呢？因为软件开发人员已经把XML当成编程语言来用了，可它不是图灵完备[①]的语言，所以它不适合干这些事。希望XML能在你的项目中得其所哉，只是用来交换数据。

本节重点是XML，一种常用的交换数据格式。尽管Java语言的核心（通过JAXB和JAXP）以及浩浩荡荡的第三方类库（XStream、Xerces、Xalan等）组成了庞大的XML处理大军，但选哪个方案经常让人难以抉择，并且采用相应方案的Java代码会变得非常冗长。

本节会带你用Groovy创建XML，并告诉你如何把XML解析为GroovyBean。

1. 创建XML

用Groovy构建XML文档非常简单，比如person：

① 对于一种语言来说，如果是图灵完备的，那它至少必须能做条件分支判断，并能修改内存数据。

```
<person id='2'>
  <name>Gweneth</name>
  <age>1</age>
</person>
```

Groovy能用内置的MarkupBuilder产生这个XML。产生personXML记录的代码如代码清单8-8所示：

代码清单8-8　产生简单的XML

```
def writer = new StringWriter()
def xml = new groovy.xml.MarkupBuilder(writer)
xml.person(id:2) {
  name 'Gweneth'
  age 1
}
println writer.toString()
```

注意看person的起始元素（属性id设置为2）创建起来多么简单，根本不用定义Person对象。Groovy不会强迫你显式地弄一个GroovyBean来支撑XML的创建，再一次节省了时间和精力。

代码清单8-8中的例子相当简单。你可以多做些试验，把输出类型StringWriter改掉，并且可以尝试用不同的构建器，比如groovy.json.JsonBuilder()，即刻创建JSON①。在处理更复杂的XML结构时，命名空间和其他特定构造的处理上也有额外的辅助方法。

你可能还希望执行反向操作，读取XML并把它解析成GroovyBean。

2. 解析XML

Groovy有几种解析XML输入的办法。表8-3列出了其中三个方法，这是从Groovy的官方文档（http://docs.codehaus.org/display/GROOVY/Processing+XML）中拿过来的。

表8-3 Groovy XML解析技术

方　　法	描　　述
XMLParser	支持XML文档的GPath表达式
XMLSlurper	跟XMLParser类似，但以懒加载的方式工作
DOMCategory	用一些语法支持DOM的底层解析

这三个用起来都很简单，但这一节我们主要关心XMLParser的用法。

注意　GPath是一种表达式语言。Groovy文档（http://groovy.codehaus.org/GPath）中有它的全部内容。

我们把代码清单8-8中产生的那个表示“Gweneth”（人名）的XML拿过来，并把它解析到一个GroovyBean Person中，如代码清单8-9所示。

① 关于这一问题，Dustin在他的博客Inspired by Actual Events（http://marxsoftware.blogspot.com/）上有一篇很棒的文章，标题是“Groovy 1.8 Introduces Groovy to JSON”。

代码清单8-9 用XMLParser解析XML

```
class XmlExample {
  static def PERSON =
  """
  <person id='2'>
    <name>Gweneth</name>
    <age>1</age>
  </person>
  """
}                                                  ❶ XML作为Groovy源码
class Person {def id; def name; def age}           Groovy中的Person定义

def xmlPerson = new XmlParser().
                    parseText(XmlExample.PERSON)   ❷ 读取XML

Person p = new Person(id: xmlPerson.@id,
                    name: xmlPerson.name.text(),
                     age: xmlPerson.age.text())    ❸ 填入GroovyBean Person中

println "${p.id}, ${p.name}, ${p.age}"
```

我们一开始抄了点儿近路，把XML文档直接放在代码中了，这样它就会出现在CLASSPATH中❶。真正的第一步是用XMLParser中的parseText()方法读取XML数据❷。然后创建新的Person对象，给它赋值❸，最后输出Person，以便你能用肉眼检查一下。

我们对Groovy的介绍到此就完成了。现在，你可能觉得心里痒痒的，想在自己的Java项目里使用一些Groovy特性！下一节，我们会带你看看Java如何跟Groovy互操作。由此你将迈出作为优秀Java开发者的重要一步：成为一名JVM多语言程序员。

8.5 Groovy 与 Java 的合作

这一节很短，但不要低估它的重要性！如果你是按顺序看到这里的，这里就是你要跳的龙门，跳过去你就不是只在JVM上做Java开发的人了。JVM上有多种语言作为Java的补充，优秀的Java开发者要具备使用它们的能力，Groovy是个很好的起点！

首先，你会重温一下从Groovy中调用Java是多么简单。之后你会看到Java与Groovy交互的三种常用途径，使用GroovyShell、GroovyClassLoader和GroovyScriptEngine。

我们先来重温一下Groovy里怎么调用Java。

8.5.1 从Groovy调用Java

还记得吗？我们说过从Groovy调用Java很简单，你只要把JAR放到CLASSPATH中，然后用标准的import语句就行了。这儿有个例子，引入流行的Joda日期时间类库中org.joda.time包里的类[1]：

```
import org.joda.time.*;
```

[1] 在Java 8发布之前，实际上Joda一直都不是Java的标准日期时间类库。

可以跟在Java中一样使用这些类。下面的代码会输出当前月份的数值表示。

```
DateTime dt = new DateTime()
int month = dt.getMonthOfYear()
println month
```

哦，肯定要比这个更复杂点儿吧？

阿克巴上将："陷阱！"①

开个玩笑，这里没什么陷阱！真的就这么简单，那我们是不是应该看看更困难的情况？从Java调用Groovy并得到有意义的结果还是有点儿技术含量的。

8.5.2　从Java调用Groovy

从Java程序调用Groovy需要把Groovy及其相关的JAR放到这个程序的CLASSPATH下，因为它们都是运行时依赖项。

> **提示**　只需要把GROOVY_HOME/embeddable/groovy-all-1.8.6.jar文件放到CLASSPATH中。

下面是几种从Java调用Groovy代码的办法：

- 使用Bean Scripting Framework（BSF），即JSR 223；
- 使用GroovyShell；
- 使用GroovyClassLoader；
- 使用GroovyScriptEngine；
- 使用嵌入式Groovy控制台。

我们在这一节重点讨论最常用的办法（GroovyShell、GroovyClassLoader和Groovy-ScriptEngine）。先从最简单的GroovyShell开始。

1. GroovyShell

在临时性快速调用Groovy并计算表达式或类似于脚本的代码时，可以用GroovyShell。比如说，有些开发人员可能更喜欢用Groovy做数值处理，就可以调用GroovyShell执行一些数学计算。代码清单8-10会返回用Groovy的数值相加得到的结果10.4。

代码清单8-10　在Java中用GroovyShell执行Groovy代码

```
import groovy.lang.GroovyShell;
import groovy.lang.Binding;
import java.math.BigDecimal;

public class UseGroovyShell {

  public static void main(String[] args) {
```

① 星战迷对这句在网上广为流传的话应该不会感到陌生。

```
    Binding binding = new Binding();
    binding.setVariable("x", 2.4);
    binding.setVariable("y", 8);
    GroovyShell shell = new GroovyShell(binding);      ⊲ 设置shell上的binding
    Object value = shell.evaluate("x + y");            ⊲ 计算并返回表达式
    assert value.equals(new BigDecimal(10.4));
  }
}
```

用GroovyShell只能应付快速执行小段Groovy代码的情况，如果要与一个完整的Groovy类交互，该怎么办呢？这时可以用GroovyClassLoader。

2. **GroovyClassLoader**

从开发人员的角度看，GroovyClassLoader的表现很像Java的ClassLoader。找到类和想要调用的方法，然后调用就行了。

下面的代码中有一个简单的CalculateMax类，其中有个getMax方法，会使用Groovy内置的max函数。要在Java里通过GroovyClassLoader运行这个方法，需要用下面的代码创建一个Groovy文件（CalculateMax.groovy）：

```
class CalculateMax {
  def Integer getMax(List values) {
    values.max();
  }
}
```

现在我们有了要执行的Groovy脚本，可以从Java调用它了。在代码清单8-11中，从Java调用CalculateMax getMax函数，返回了传入参数中的最大值10。

8

代码清单8-11 在Java中用GroovyClassLoader执行Groovy代码

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import groovy.lang.GroovyClassLoader;
import groovy.lang.GroovyObject;
import org.codehaus.groovy.control.CompilationFailedException;

public class UseGroovyClassLoader {

  public static void main(String[] args) {
    GroovyClassLoader loader = new GroovyClassLoader();      ⊲ 准备GroovyClassLoader

    try {
      Class<?> groovyClass = loader.parseClass(
                    new File("CalculateMax.groovy"));        ⊲ 得到Groovy类

      GroovyObject groovyObject = (GroovyObject)
                         groovyClass.newInstance();          ⊲ 得到Groovy类的实例

      ArrayList<Integer> numbers = new ArrayList<>();        ⊲ 准备参数
      numbers.add(new Integer(1));
      numbers.add(new Integer(10));
      Object[] arguments = {numbers};
```

```
      Object value =
        groovyObject.invokeMethod("getMax", arguments);     ← 调用Groovy方法
      assert value.equals(new Integer(10));
    }
    catch (CompilationFailedException | IOException | InstantiationException
            | IllegalAccessException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

这种技术在调用几个Groovy实用类时可能会有用。但如果要用大量的Groovy代码，我们推荐使用完整的GroovyScriptEngine。

3. GroovyScriptEngine

使用GroovyScriptEngine要指明Groovy代码的URL或所在目录。Groovy脚本引擎会加载那些脚本，并在必要时进行编译，包括其中的依赖脚本。比如说你修改了脚本B，而脚本A依赖于B，则引擎会全重新编译它们。

假设有一个Groovy脚本（Hello.groovy）定义了一个简单的“Hello”语句，后面跟着一个名字（要从Java应用程序中传入的参数）。

```
def helloStatement = "Hello ${name}"
```

然后Java程序会通过GroovyScriptEngine使用Hello.groovy，并输出一句问候，如代码清单8-12所示：

代码清单8-12　在Java中用GroovyScriptEngine执行Groovy代码

```
import groovy.lang.Binding;
import groovy.util.GroovyScriptEngine;
import groovy.util.ResourceException;
import groovy.util.ScriptException;
import java.io.IOException;

public class UseGroovyScriptEngine {
  public static void main(String[] args)
  {
    try {
      String[] roots = new String[] {"/src/main/groovy"};     ← 设置根目录
      GroovyScriptEngine gse =
                         new GroovyScriptEngine (roots);       ← 初始化引擎

      Binding binding = new Binding();
      binding.setVariable("name", "Gweneth");

      Object output = gse.run("Hello.groovy", binding);       ← 运行脚本
      assert output.equals("Hello Gweneth");
    }
    catch (IOException | ResourceException | ScriptException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

GroovyScriptEngine监控之下的任何Groovy脚本都可能被程序员一时兴起改掉。比如说，将Hello.groovy改成这样：

```
def helloStatement = "Hello ${name}, it's Groovy baby, yeah!"
```

这段Java代码下次再运行时，它就会用这个新的，更长的消息。这样Java应用程序就具备了以前根本不可能出现的动态灵活性。这在某些情况下简直是无价之宝，比如调试生产环境下的代码，在运行时修改系统属性，还有很多……

至此，对Groovy的介绍真要结束了。我们已经走了很远了！

8.6 小结

Groovy有多种引人注目的特性，这使它成为一门可以和Java共用的出色语言。你可以用和Java非常相近的语法，也可以用更精简的代码实现相同的逻辑。这种精简并不以牺牲可读性为代价，而且Java开发人员在采用跟集合、null引用处理和GroovyBean相关的新语法时不存在什么困难。然而，Groovy给Java开发人员设了几个陷阱，但你已经搞定了大多数情况，希望你能带领同事走进这片新大陆。

很多Java开发人员都对Groovy中的几个语言特性感到眼馋，希望有朝一日Java语言中也能有这些特性。其中最难掌握、也最强大的就是函数字面值，它是一种能在集合上轻松进行操作的强大编程技术（跟其他技术一起）。当然，集合享受的是一等公民的待遇，你能用更短小易用的语法来创建、修改和操作它们。

大多数Java开发人员都要在Java程序里生成或解析XML，对此Groovy也能助你一臂之力，它能用内置的XML支持帮你挑起大部分重担。

借助各种技术把Java代码和Groovy代码集成在一起解决编程问题，你已经向多语言程序员迈出了一步。

我们的Groovy旅程还没有结束。在第13章讨论快速Web开发时，还有更多的Groovy特性等着我们去使用和探索。

接下来，我们请出Scala，另外一门已在业内造成小轰动的JVM语言。Scala既是面向对象语言，也是函数式语言，要解决现代编程中进退两难的问题，Scala是值得一看的语言。

第 9 章

Scala：简约而不简单

9

本章内容
- ❑ Scala不是Java
- ❑ Scala语法及更加函数化的风格
- ❑ match表达式与模式
- ❑ Scala的类型系统和集合
- ❑ Scala并发之actor

Scala是出自学术界和编程语言研究社区的语言。由于其强大的类型系统和先进特性，又有精英团队证明了其价值，它已经赢得了一定数量开发者的青睐。

现在Scala里有很多有意思的地方，但要评判它能否完全渗入Java生态系统，以及能否挑战Java语言的霸主地位，还为时尚早。

最合理的预测是Scala会被更多的团队接受，并最终被项目采纳。在接下来的三四年中，我们预计会有大量开发人员在项目中见到Scala的身影。也就是说作为优秀的Java开发者，你应该了解它，能判断出它是否适用于自己的项目。

例子 金融风险模拟应用可能需要采用Scala新颖的面向对象方式、类型推断、灵活的语法、新的集合类（包括自然的函数式编程风格，比如映射/过滤器惯用语），以及基于actor的并发模型。

对于Java开发人员来说，刚开始接触Scala时最应该牢记于心的是：Scala不是Java。

这看起来似乎是显而易见的。毕竟，每种语言都不同，Scala当然也和Java不一样。但我们在第7章提到过，有些语言非常相似。第8章介绍Groovy时也在强调它和Java相似的地方。希望这些内容在你首次探索Groovy这门非Java JVM语言时能够对你有所帮助。

本章我们想做点不一样的事情，先突出Scala中一些非常独特的语言特性。我们喜欢把这比喻成“Scala的宜居之所”，告诉你怎么写Scala代码才不会像是从Java翻译过来的。之后我们会阐述项目问题，搞明白Scala是不是适用于你的项目。然后，我们看一些Scala在语法上的创新，这些创新让Scala代码变得即简洁又漂亮。接下来是Scala处理面向对象的方式，然后是一节介绍集合

和数据结构的内容。最后收尾的一节是关于Scala并发和它强大的actor模型的内容。

在讨论Scala的独特性时，我们会一并解释它的语法（以及其他必要的概念）。跟Java比，Scala是一门相当庞大的语言，需要掌握的基础概念和语法点更多。这就是说你要做好心理准备，随着接触到的Scala代码越来越多，你需要自己花时间和精力去更多地探索这门语言。

我们先来大体看一下后面会遇到的一些主题。这能帮你熟悉Scala不同的语法和思维方式，并帮你打下基础，以便更好地学习新知识。

9.1 走马观花 Scala

下面是我们准备展示的主要内容：

- Scala语言的精炼，包括类型推断的能力；
- match表达式，以及模式和case类等相关概念；
- Scala的并发，采用消息和actor机制，而不是像Java代码那样用老旧的锁机制。

这些不是Scala的全部内容，只掌握它们也不可能让你变成Scala开发高手。它们是用来吊你胃口的，只是给你几个具体示例表明Scala可能适用于哪些场合。要走得更远，就得做更深入的探索。你可以找些在线资源，也可以找本完整讲述Scala的书，比如Joshua Suereth的*Scala in Depth*（Manning，2012）。

我们要解释的第一个特性，也是Scala跟Java最重要的差别，就是它语法上的精炼性，我们就直奔主题吧。

9.1.1 简约的Scala

Scala是采用静态类型系统的编译型语言。也就是说Scala代码应该和Java代码一样详细。可Scala偏偏很精炼，它太精炼了，看起来简直和脚本语言一样。因此Scala开发人员更加快速和高效，写代码的速度几乎可以跟用动态语言编程媲美了。

我们来看一些非常简单的代码，了解一下Scala的构造方法和类。比如要写一个简单的现金流模型类。需要用户提供两项信息：现金流的额度和货币。用Scala应该这样写：

```
class CashFlow(amt : Double, curr : String) {
  def amount() = amt
  def currency() = curr
}
```

这个类只有四行（其中一行还是用来结束的右括号）。不管怎样，它有获取方法（但没有设置方法）作为参数，还有一个单例构造方法。跟Java比起来，这简直太划算了（就这么几行代码）。请看相应的Java代码：

```
public class CashFlow {
  private final double amt;
  private final String curr;

  public CashFlow(double amt, String curr) {
```

```
    this.amt = amt;
    this.curr = curr;
  }

  public double getAmt() {
    return amt;
  }

  public String getCurr() {
    return curr;
  }
}
```

跟Scala相比，Java代码中的重复信息太多了，就是这种重复导致了Java代码的冗长。

选择Scala，让开发人员尽量减少重复信息的输入，IDE的界面中就可以显示更多内容。面对稍微复杂点的逻辑时，开发人员就能见到更多代码，因此也有望能掌握理解它所需的更多线索。

要不要省1500美元？

CashFlow类的Scala版长度几乎比Java版短75%。据估计，一行代码每年的成本是32美元。如果我们假定这段代码的生命期是5年，那在这个项目的生命期内，Scala版代码的维护成本就会比Java代码少花1500美元。

既然说到这儿了，我们就来看看第一个例子中展示的语法点。

- 类的定义（就它的参数而言）和类的构造方法是同一个东西。Scala中可以有其他的“辅助构造方法”，稍后就会谈到。
- 类默认是公开的，所以没必要加上public关键字。
- 方法的返回类型是通过类型推断确定的，但要在定义方法的def从句中用等号告诉编译器做类型推断。
- 如果方法体只是一条语句（或表达式），那就没必要用大括号括起来。
- Scala不像Java一样有原始类型。数字类型也是对象。

Scala的精炼不止体现在这些方面。甚至像HelloWorld这样简单的经典程序中都有所体现：

```
object HelloWorld {
  def main(args : Array[String]) {
    val hello = "Hello World!"

    println(hello)
  }
}
```

即便在这个最基本的例子中，也有几个帮我们去除套路化代码的特性。

- 关键字object告诉Scala编译器这个类是单例类。
- 调用println()没必要说明完整路径（感谢默认引入）。
- 没必要在main()方法前指明关键字public和static。
- 不必声明hello的类型，编译器会自己找出来。

- 不必声明`main()`的返回类型，编译器会自动设为`Unit`（等价于Java中的`void`）。

这个例子中还有些相关语法需要注意一下。

- 跟Java和Groovy不一样，变量的类型在变量名之后。
- Scala用方括号来表示泛型，所以类型参数的表示方法是`Array[String]`，而不是`String[]`。
- `Array`是纯正的泛型。
- 集合类型必须指明泛型（不能像Java那样声明生类型[①]）。
- 分号绝对是可选的。
- `val`就相当于Java中的`final`变量，用于声明一个不可变变量。
- Scala应用程序的初始入口总是在`object`中。

在后续几节中，我们会详细解释这些语法是如何工作的，并且我们还会再选几个让你更省手指头的Scala创新介绍一下。我们也会讨论Scala的函数式编程，它对于编写精炼的代码非常有帮助。现在，我们先来讨论一个强大的Scala"本地"特性。

9.1.2 match表达式

Scala有一种非常强大的结构：`match`表达式。最简单的`match`用法跟Java的`switch`差不多，但`match`的表达力要强得多。`match`表达式的形式取决于`case`从句中的表达式结构。Scala调用不同类型的`case`从句模式，但要注意，这些所谓的模式跟正则表达式里的"模式"是截然不同的（尽管在`match`表达式里也可以用正则表达式模式）。

先看一个熟悉的例子。1.3.1节那个带字符串的`swtich`被翻译成了Scala代码，请看：

```
var frenchDayOfWeek = args(0) match {
  case "Sunday"    => "Dimanche"
  case "Monday"    => "Lundi"
  case "Tuesday"   => "Mardi"
  case "Wednesday" => "Mercredi"
  case "Thursday"  => "Jeudi"
  case "Friday"    => "Vendredi"
  case "Saturday"  => "Samedi"
  case _           => "Error: '"+ args(0) +"' is not a day of the week"
}
println(frenchDayOfWeek)
```

我们在这个例子中只用到了两种最基本的模式：用来确定是周几的常量模式和处理默认情况的_模式，后面我们还会遇到其他模式。

从语言的纯粹性来看，可以说Scala的语法比Java更清晰，也更正规，至少从下面这两点来看是这样的：

① 生类型（raw type）是指不带类型参数的泛型类或接口。比如泛型类`Box<T>`，创建它的参数化类型时要指明类型参数的真实类型：`Box<Integer> intBox = new Box<>();`。如果忽略了类型参数，`Box rawBox = new Box();`则是创建了一个生类型。——译者注

- 默认case不需要另外一个不同的关键字；
- 单个case不会像Java中那样进入下一个case，所以也不需要break。

这个例子中的其他语法点如下所示。

- 关键字var用来声明一个可变（非final）变量。没有必要尽量不要用它，但有时候确实需要它。
- 数组用圆括号访问，比如args(0)是指main()的第一个参数。
- 总应该包括默认case。如果Scala在运行时在所有case中都找不到匹配项，就会抛出MatchError。这绝不是你想看到的。
- Scala支持间接方法调用，所以可以把args(0).match({ ... })写成args(0) match { ... }。

到目前为止一切都好。match看起来就像稍微简洁些的switch。但这只是它众多模式中最像Java的。Scala中有大量使用不同模式的语言结构。比如说，有一种类型化模式，对于处理类型不确定的数据很有用，不用像Java那样弄一堆乱糟糟的类型转换或instanceof测试：

```
def storageSize(obj: Any) = obj match {
  case s: String => s.length
  case i: Int    => 4
  case _         => -1
}
```

这个极其简单的方法以一个Any类型(即未知类型)的值为参数,然后用模式分别处理String和Int类型的值。每个case都给要处理的值绑定了一个临时别名，以便必要时可以调用其中的方法。

在Scala的异常处理代码中有一个跟变量模式非常相似的语法形式。下面是一段改编自第11章ScalaTest框架的类加载代码：

```
def getReporter(repClassName: String, loader: ClassLoader): Reporter = {
  try {
    val reporterCl: java.lang.Class[_] = loader.loadClass(repClassName)
    reporterCl.newInstance.asInstanceOf[Reporter]
  }
  catch {
    case e: ClassNotFoundException => {
      val msg = "Can't load reporter class"
      val iae = new IllegalArgumentException(msg)
      iae.initCause(e)
      throw iae
    }
    case e: InstantiationException => {
      val msg = "Can't instantiate Reporter"
      val iae = new IllegalArgumentException(msg)
      iae.initCause(e)
      throw iae
    }
...
  }
}
```

在getReporter()中，要加载一个定制的report类（通过反射），以便在运行测试集时输出报告。在类加载和实例化过程中很多事都可能出错，所以要有个try-catch块来保护程序执行。

catch块起到的作用就跟在异常类型上放match表达式类似。case类的这种思路还可以进一步延伸，接下来我们就来讨论这个。

9.1.3 case类

match表达式的最强用法之一就是跟case类(可以看成是枚举概念面向对象的扩展)相结合。我们来看一个温度过高发出报警信号的例子：

```
case class TemperatureAlarm(temp : Double)
```

单这一行代码就可以定义一个绝对有效的case类。在Java中相应的类大概应该是这样子：

```
public class TemperatureAlarm {
  private final double temp;
  public TemperatureAlarm(double temp) {
    this.temp = temp;
  }

  public double getTemp() {
    return temp;
  }

  @Override
  public String toString() {
    return "TemperatureAlarm [temp=" + temp + "]";
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    long temp;
    temp = Double.doubleToLongBits(this.temp);
    result = prime * result + (int) (temp ^ (temp >>> 32));
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    TemperatureAlarm other = (TemperatureAlarm) obj;
    if (Double.doubleToLongBits(temp) !=
        Double.doubleToLongBits(other.temp))
        return false;
    return true;
  }
}
```

只需加个case关键字就可以让Scala编译器生成这些额外的方法。它还会生成很多额外的架子方法。大多数情况下，开发人员都不会直接使用这些方法。它们是为某些Scala特性提供运行时支持的——能以“自然的Scala”方式使用case类。

创建case类实例不需要关键字new，像这样：

```
val alarm = TemperatureAlarm(99.9)
```

这进一步强化了case类是类似于“参数化枚举类型”或某种形式的值类型的观点。

Scala中的相等

Scala认为Java用==表示“引用相等”是个错误。所以在Scala中，==和.equals()是一样的。如果需要判断引用相等，可以用===。case类的.equals()方法只有在两个实例的所有参数值都一样时才会返回true。

case类跟构造器模式非常合，请看：

```
def ctorMatchExample(sthg : AnyRef) = {
  val msg = sthg match {
    case Heartbeat => 0
    case TemperatureAlarm(temp) => "Tripped at temp "+ temp
    case _ => "No match"
  }
  println(msg)
}
```

我们去看看Scala观光之旅的最后一站：基于actor的并发结构。

9.1.4　actor

Scala选择用actor机制来实现并发编程。它们提供了一个异步并发模型，通过在代码单元间传递消息实现并发。很多开发人员都发现这种并发模型比Java提供的基于锁机制、默认共享的并发模型易用（不过Scala的底层模型也是JMM）。

来看个例子。假设我们在第4章遇到的兽医需要监控诊所里动物的健康状况（尤其是体温）。按我们的想法，温度感应器应该会将它们的读数消息发送给中心监控软件。

在Scala中，我们可以用一个actor类TemperatureMonitor对这种设置建模。应该有两种不同的消息：一种是标准的“心跳”消息，一种是TemperatureAlarm消息。第二种消息会带一个参数，表明那个警报器的温度超出了限值。代码清单9-1中列出了这些类的代码。

代码清单9-1　与actor的简单通信

```
case object Heartbeat
case class TemperatureAlarm(temp : Double)

import scala.actors._

class TemperatureMonitor extends Actor {
```

```
    var tripped : Boolean = false
    var tripTemp : Double = 0.0

    def act() = {
      while (true) {
        receive {
          case Heartbeat => 0
          case TemperatureAlarm(temp) =>
            tripped = true
            tripTemp = temp
          case _ => println("No match")
        }
      }
    }
  }
```


监控actor会对三种不同的case做出响应（通过receive）。第一个是心跳消息，告诉你一切正常。因为这个case类没有参数，所以技术上来说它是一个单例实例，可以按case对象引用。actor在收到心跳消息时什么也不用做。

如果收到TemperatureAlarm消息，actor会保存警报器上的温度值。你应该想象得出，兽医有另外的代码定期检查TemperatureMonitor actor，看有没有警报被触发。

最后还有个default case。这是为了确保有任何不期而至的消息溜进actor环境时能被捕获到。如果没有这个一切全包的case，actor如果看到不认识的消息类型就会抛出异常。我们在本章的最后还会再次讨论actor的更多细节，但Scala的并发是个非常大的主题，而且在这本书里我们也不想让你浅尝辄止。

我们快速浏览了Scala的一些亮点。希望其中的某些特性已经燃起了你的兴趣之火。在下一节，我们会花点时间聊聊你可能会（也可能不会）在自己的项目中选择使用Scala的原因。

9.2 Scala 能用在我的项目中吗

在Java项目里增加一门语言总该有正当的理由和根据。在这一节，我们希望你想想那些理由，以及如何把它们应用到你的项目中。

一开始我们会快速比较一下Scala跟Java，然后看看什么时候，以及如何开始使用Scala。为了让这一篇幅较短的章节更完整，我们会看一些示警信号，当出现这些信号时，Scala可能不是最适合你项目的语言。

9.2.1 Scala和Java的比较

我们在表9-1中对这两种语言的主要差异做了汇总。语言的“表皮层”是指该语言关键字的数量和开发人员用它干活必须掌握的独立语言结构的数量。

这些差异是Scala得到Java开发人员青睐，可以把它当做某些项目或组件的备选语言的部分原因。接下来我们会给出把Scala引入项目的更多细节。

表9-1　比较Scala和Java

特　性	Java	Scala
类型系统	静态的、非常繁琐	静态的，但使用了大量类型推断
多语言金字塔层级	稳定	稳定、动态
并发模型	基于锁机制	基于actor机制
函数式编程	需要严格遵守的特殊编码方式，不自然	内置支持、语言的一部分（纯天然）
表皮层	小型/中型	大型/超大型
语法风格	简单、常规、比较繁琐	灵活、精炼、很多特殊情况

9.2.2　何时以及如何开始使用Scala

我们在第7章讨论过，在已有项目中引入新语言时，最好从风险比较低的区域开始。ScalaTest测试框架（见第11章）就是这样的低风险区。如果Scala实验不顺利，那所有的成本就是开发人员浪费的时间（这些单元测试有可能变成普通的JUnit测试）。

一般来说，适合在项目中引入Scala的组件应该基本满足下面这些条件：

- 你有信心评估所需的工作量；
- 问题域边界明确，定义清晰；
- 需求说明正确；
- 与其他组件的互操作性需求已知；
- 确定了愿意学习新语言的开发人员。

经过深思熟虑选定合适区域后，你就可以开始实现自己的第一个Scala组件了。下面有些指导原则，事实证明它们能让初始组件按部就班地进行：

- 以快速扣杀开始；
- 尽早跟已有的Java组件测试交互操作；
- 有定义扣杀成功或失败的入门标准（基于需求）；
- 如果扣杀失败，要有B计划；
- 在预算中为新组件留出额外的重构时间（用新语言编写的第一个项目欠下的技术债几乎肯定会比用团队已经熟悉的语言编写来得高）。

在评估Scala时，另外一个应该考虑的是检查那些明显让Scala对项目来说不太理想的迹象。

9.2.3　Scala可能不适合当前项目的迹象

下面这些迹象表明Scala可能并不适合你的项目。如果出现了其中一个或多个迹象，你应该慎重考虑引入Scala的时机是否恰当。如果超过两个，那基本就没什么戏了。

- 受到了业务小组和其他程序支持小组的抵制，或缺乏动力。
- 开发团队没有明显的学习Scala的动力。
- 小组中分帮结派或政治上存在巨大分歧。

- ❑ 小组中高级技术人员的支持力度不够。
- ❑ 截止日期太紧张（没时间学习新语言）。

另外一个要密切关注的因素是，团队是否分散在全球各地。如果你用来开发（或支持）Scala代码的员工分散在几个地方，那会增加Scala培训人员的成本和负担。

现在我们已经讨论过把Scala引入项目的机制了，接下来该去看看Scala的语法了。我们会重点关注让Java开发人员更轻松的特性，鼓励更加紧凑的代码，少点套路化和挥之不去的繁琐。

9.3　让代码因 Scala 重新绽放

我们在这一节会先介绍一下Scala编译器和交互环境（REPL）。然后讨论类型推断，接着是方法声明（跟你所熟悉的Java方式不太一样）。这两个特性能帮你减少大量的套路化代码，从而提高生产力。

我们会谈到Scala的代码封包方式和更强大的`import`语句，然后详细讲解一下Scala中的循环和控制结构。这些特性植根于跟Java差异巨大的编程传统，所以我们会借此机会讨论一下Scala的函数式编程，包括函数式的循环结构、`match`表达式和函数字面值。

看过这些之后，本章剩下的大部分内容对你来说都没什么问题了，你可以自信地说自己有能力成为一名Scala程序员了。来吧，现在我们就开始讨论编译器和内置的交互环境。

9.3.1　使用编译器和REPL

Scala是编译型语言，所以执行Scala程序通常要把它们先编译成.class文件，然后在类路径上有scala-library.jar（Scala运行时类库）的JVM环境中执行。

如果你还没装Scala，请在继续阅读之前参见附录C，了解如何安装Scala。样例程序（9.1.1中的HelloWorld）可以用`scalac HelloWorld.scala`编译（如果你正好在HelloWorld.scala文件所在的目录中）。

一旦得到.class文件，就可以用命令`scala HelloWorld`执行它了。这个命令会启动带着Scala运行时环境的JVM，然后进入类文件指定的main方法。

除了编译和运行，Scala还有个内置的交互环境，有点像第8章讲的Groovy控制台。但不像Groovy，Scala是在命令行环境里实现的。这就是说在典型的Unix/Linux环境（`Path`设置正确）中，你可以敲入`scala`，它就会在终端窗口内打开，而不会再弹出一个新窗口。

注意　这类交互环境有时被称为读入—计算—输出（Read-Eval-Print）循环，或简称为REPL。这在动态语言中很常见。在REPL环境中，前面输入的那些行的计算结果还在，在后面的表达式和计算中还可以用。在本章的剩余部分，我们偶尔会用REPL环境来演示Scala语法。

现在我们开始讨论下一个大特性：Scala的高级类型推断。

9

9.3.2　类型推断

在读前面的代码时你可能已经注意到了，我们在声明变量hello为val时，没有指明它是什么类型。因为它很“明显”是个字符串。表面上来看这有点像Groovy，变量没有类型（Groovy是动态类型语言），但其实Scala代码中所发生的事情完全不同。

Scala是静态类型语言（所以变量确实有明确的类型），但它的编译器能分析源码，并且一般都能根据上下文推断出应该是什么类型。如果Scala自己能确定是什么类型，就不用你亲自告诉它了。

这就是类型推断，我们已经提过好几次了。Scala在这方面的能力非常突出—以致于开发人员经常在行云流水一样的代码中忘记静态类型。这经常让Scala更有动态语言的“感觉”。

Java中的类型推断

Java也有类型推断的能力，虽然有限，但确实有。最明显的例子就是我们在第1章见到的泛型钻石语法。Java的类型推断通常是用在赋值语句等号右边的值上。Scala通常是推断变量而不是值的类型，但它的确也能推断值的类型。

你已经见过其中最简单的例子了：关键字var和val，Scala根据赋给变量的值来推断它们的类型。Scala类型推断的另一个重要应用是方法声明。我们来看个例子（Scala的AnyRef就是Java中的Object）：

```
def len(obj : AnyRef) = {
  obj.toString.length
}
```

这是一个类型推断的方法。通过检查它返回代码中的java.lang.String#length的类型（int），编译器知道这个方法要返回Int类型的值。注意，这个方法没有显式指定返回类型，我们也不需要用return关键字。实际上，如果你放了一个显式的return在这里，像这样：

```
def len(obj : AnyRef) = {
  return obj.toString.length
}
```

会得到一个编译时错误：

```
error: method len has return statement; needs result type
      return obj.toString.length
      ^
```

如果你连def中的=也省略了，编译器会假定这个方法会返回Unit（就跟Java里返回void一样）。

除了前面那些限制，还有两个类型推断受限的区域：

- **方法声明中参数的类型**——传给方法的参数必须指定类型；
- **递归函数**——Scala编译器不能推断递归函数的返回类型。

关于Scala的方法，我们讨论的东西已经不少了，但还算不上系统化的讨论，所以我们来巩固一下你已经学过的东西。

9.3.3 方法

你已经见过怎么用def关键字定义方法了。随着你对Scala越来越熟悉，关于Scala的方法，还有些你应该知道的重要事实。

- Scala没有static关键字。跟Java中的static方法对应的方法必须放在Scala的object（单例）结构中。稍后我们会向你介绍相关概念：伴生对象。
- 跟Groovy（或Clojure）相比，Scala语言的运行时要重得多。Scala类中可能会有很多由平台自动生成的额外方法。
- 方法调用是Scala的核心概念。在Scala中没有Java中那种意义的操作符。
- 对于哪些字符可以出现在方法的名称中，Scala比Java更灵活。特别是那些在其他语言中作为操作符的字符，在Scala中可能是合法的方法名（比如加号+）。

间接方法调用（前面讲过）中有Scala把方法调用和操作符合并到一起的线索。举个例子，比如要把两个整型相加。在Java中，应该是写一个a+b这样的表达式。在Scala中你也可以这样写，但不止这样，还可以写成a.+(b)。换句话说，你调用了a上的+()方法，并把b作为参数传给它。这就是Scala不再把操作符当做一个独立概念的秘密。

注意 你可能已经注意到了，a.+(b)是在a上调用方法。但原始类型的变量a怎么会有方法呢？9.4节会给出完整的解释。但现在，你只要知道Scala的类型系统认为所有东西都是对象，所以你可以在任何东西上调用方法，即便是Java里的原始类型变量也行。

你已经见过一个用def关键字声明方法的例子了。我们再来看一个例子，一个实现阶乘函数的简单递归方法：

```
def fact(base : Int) : Int = {
  if (base <= 0)
    return 1
  else
    return base * fact(base - 1)
}
```

对于所有负数，这个函数都返回1，这算是作弊吧。实际上，负数的阶乘是不存在的，但大家都是朋友嘛。它看起来有点像Java：有返回类型（Int），并用return关键字表明把哪个值交回给调用者。唯一需要注意的就是在函数体代码块定义之前额外符号=。

Scala中还有另外一个Java中没有概念：局部函数。它是在另外一个函数内部（并且仅在这一作用域内有效）定义的函数。如果开发人员想要一个辅助函数，又不想把实现细节暴露给外部，这是一个简单的办法。在Java中除了用private方法之外别无选择，但这个函数对于同一类的其他方法都是可见的。但在Scala中，你只要这样写就行了：

```
def fact2(base : Int) : Int = {

  def factHelper(n : Int) : Int = {
    return fact2(n-1)
  }

  if (base <= 0)
    return 1
  else
    return base * factHelper(base)
}
```

factHelper()在fact2()的封闭作用域之外绝对是不可见的。

接下来，我们去看看Scala如何处理代码的组织和导入。

9.3.4　导入

Scala对包的使用跟Java一样，关键字也一样，分别是package和import。Scala可以毫无障碍地导入和使用Java的包和类。Scala的var或val变量可以引用任何Java类的实例，不需要任何特殊的语法或处理：

```
import java.io.File
import java.net._
import scala.collection.{Map, Seq}
import java.util.{Date => UDate}
```

头两行代码跟Java里的标准导入和通配符导入一样。第三行用一行导入一个包里的多个类。最后一行在导入时指定了类的别名（避免缩写冲突出现）。

跟Java不一样，Scala中的import可以出现在代码中的任何位置（不仅限于文件顶部），这样你就可以把import当做文件的一部分分离出来。Scala也有默认导入，即所有.scala文件默认都会导入scala._。这里有很多有用的函数，包括我们已经讨论过的一些，比如println。对于所有默认导入的完整细节，请参见www.scala-lang.org/上的API文档。

我们接下来讨论怎么控制Scala程序的执行流。这可能和你熟悉的Java跟Groovy有些差异。

9.3.5　循环和控制结构

Scala在控制和循环结构上引入了几个有点绕的创新。在我们向你介绍这些不熟悉的形式之前，先来看几个老朋友，比如标准的while循环：

```
var counter = 1
while (counter <= 10) {
  println("." * counter)
  counter = counter + 1
}
```

还有do-while形式：

```
var counter = 1
do {
  println("." * counter)
  counter = counter + 1
} while (counter <= 10)
```

另一个是基本的for循环：

```
for (i <- 1 to 10) println(i)
```

看起来都很好。但Scala更灵活，比如条件for循环：

```
for (i <- 1 to 10; if i %2 == 0) println(i)
```

还能在多个变量上循环，比如：

```
for (x <- 1 to 5; y <- 1 to x)
  println(" " * (x - y) + x.toString * y)
```

这些多出来的形式源于Scala实现这些结构的根本性差异。Scala用函数式编程中的概念（*列表推导式*）来实现for循环。

列表推导式的一般概念是对一个列表中的元素进行转换（或过滤，比如在用条件for循环时）。这会产生一个新列表，然后在其中的每个元素上逐次运行for循环体中的代码。

甚至把要过滤的列表和for代码块分开都是有可能的，用yield关键字。比如下面这段代码：

```
val xs = for (x <- 2 to 11) yield fact(x)
for (factx <- xs) println(factx)
```

这段代码先设置新集合xs，然后用第二个for循环逐一输出其中的值。如果你需要一个创建一次、使用多次的集合，这个极其好用。

这一结构能成立是因为Scala支持函数式编程，我们接下来就去看看Scala如何实现函数式思想。

9.3.6 Scala的函数式编程

我们在7.5.2节提起过，Scala把函数当做内置的值。这就是说函数可以放进var或val中，并和其他任何值所受的对待毫无二致。这被称为*函数字面值*（*或匿名函数*），它们是Scala世界观的重要组成部分。

在Scala中写函数字面值非常简单。其中的关键是箭头=>，Scala用它来表示取得参数列表并传递到代码块中：

```
(<函数参数列表>) => { ... 作为代码块的函数体 ... }
```

我们用Scala的交互环境来演示一下。下面这个例子中定义的函数接受一个Int参数，然后乘以2：

```
scala> val doubler = (x : Int) => { 2 * x }
doubler: (Int) => Int = <function1>

scala> doubler(3)
res4: Int = 6

scala> doubler(4)
res5: Int = 8
```

注意看Scala怎么推断doubler的类型。它的类型是“接受一个Int并返回Int的函数”。这样

的类型用Java的类型系统还不能以令人完全满意的方式表示。你看，调用doubler就是用标准的调用语法。

我们把这个概念再向前推进一点。在Scala中，函数字面值只是值。并且是函数返回的值。这就是说你可以写一个生产函数的函数——接受一个值并返回一个新的函数字面值。

比如说，可以定义一个命名为adder的函数字面值。adder()能生产一个给它们的参数加上一个常量的函数：

```
scala> val adder = (n : Int) => { (x : Int) => x + n }
adder: (Int) => (Int) => Int = <function1>

scala> val plus2 = adder(2)
plus2: (Int) => Int = <function1>

scala> plus2(3)
res2: Int = 5

scala> plus2(4)
res3: Int = 6
```

看到了吧，Scala对函数字面值支持得很好。实际上，Scala代码一般都能用非常函数的世界观来编写，同时也能用更加命令式的风格编写。现在我们所做的不过是刚刚涉足Scala的函数式编程能力，但重要的是知道它们在那里。

在下一节中，我们会讨论Scala的对象模型和面向对象方式的细节。在一些重要方面，Scala的一些先进的特性使得它对面向对象的处理方式跟Java差异很大。

9.4　Scala 对象模型：相似但不同

Scala有时被称为“纯粹”的面向对象语言。也就是说所有的值都是对象，所以面向对象的概念几乎随处可见。本节一开始，我们会探索一下“一切皆对象”的后果。这个主题会很自然地引导我们去思考Scala的类型层级。

Scala的类型层级跟Java有几个重要差异，包括装箱和拆箱等Scala处理原始类型的方式。之后我们会考虑Scala的构造方法和类定义，以及它们怎么帮你少写代码。接着是关于trait（特质）的话题，然后再讨论Scala的单例、伴侣和包对象。本节最后我们会看一下怎么用case类再进一步减少套路化代码，并以一个有警示意义的Scala语法故事作为结尾。

让我们开始吧。

9.4.1　一切皆对象

Scala的观点是所有类型都是对象类型。包括Java所谓的原始类型。图9-1展示了Scala的类型继承关系，包括所有值类型（即原始类型）和引用类型，并标注了与Java中类型的对应关系。

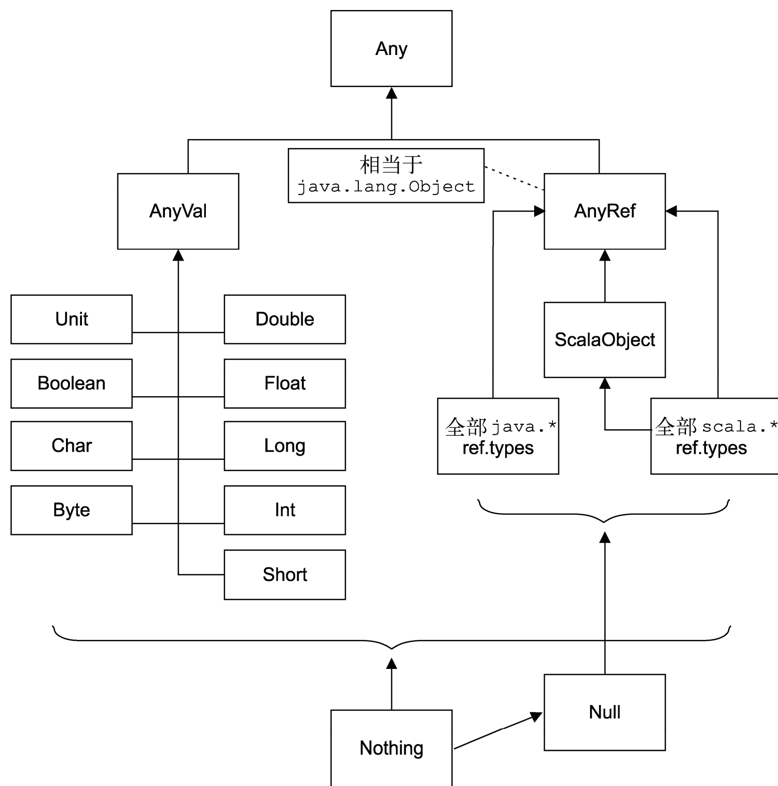


图9-1 Scala中的继承层级

从图中可以看到，`Unit`和其他值类型在Scala中都是正确的类型。`AnyRef`类相当于`java.lang.Object`。每次见到`AnyRef`，你都应该在心里把它换成`Object`。它之所以没叫`Object`，是因为Scala也要运行在.NET运行时平台上，所以它要给这个概念再起个名字。

Scala用`extends`关键字表示类的继承关系，而且它的用法跟Java很像：所有的非私有成员都会被继承下来，两种类型之间也会建立起父类/子类的关系。如果类定义中没有显式扩展其他类，则编译器会认定它是`AnyRef`的直接子类。

“一切皆对象”的原则可以解释使用中缀符号的方法调用。9.3.3节中的`obj.meth(param)`和`obj meth param`是方法调用的两种方式，其含义是一样的。现在你应该明白了，Java中的表达式`1+2`是数值原始类型和加法操作符的表达式，而Scala中与之对应的`1.+(2)`是`Scala.Int`类上的方法调用。

Scala中没有因数值的装箱操作而引起的困扰，而这在Java里很常见。请看下面的Java代码：

```
Integer one = new Integer(1);
Integer uno = new Integer(1);
System.out.println(one == uno);
```

你可能觉得很奇怪，这段代码的输出结果居然是`false`。而Scala中对数值装箱及相等判断的

方式符合我们的常识，这有以下几个好处。

- 数值类不能由构造方法实例化。它们是有效的abstract和final类（Java中不允许这种组合）。
- 得到数值类实例的唯一办法就是作为字面值。这能确保2总是同一个2。
- 判断两个值是否相等所用的==方法的定义和equals()一样，不是引用相等。
- ==不能重写，但equals()可以。
- 对于引用相等的判断，Scala中有eq方法。但一般不太会用到它。

现在我们已经讨论了Scala中一些最基本的面向对象概念，还需要再多介绍一点儿Scala的语法。最简单的就是Scala的构造方法。

9.4.2 构造方法

Scala的类必须有个主构造方法来定义该类所需的参数。此外，类还可以有额外的辅助构造方法。这些辅助构造方法都用this()表示，但它们比Java的重载构造方法限制更严格。

Scala辅助构造方法的第一条语句必须调用同一个类中的另一个构造方法（或者是主构造方法，或者是另一个辅助构造方法）。这种限制是为了把控制流引导到主构造方法上，因为它是类的唯一真正入口。也就是说，辅助构造方法的真实作用是为主构造方法提供默认参数。

请看CashFlow上的这些辅助构造方法：

```
class CashFlow(amt : Double, curr : String) {
  def this(amt : Double) = this(amt, "GBP")
  def this(curr : String) = this(0, curr)

  def amount = amt
  def currency = curr
}
```

这个例子中有个辅助函数可以只给出金额，CashFlow会假定货币是英镑。另一个辅助函数可以只给出货币，假定金额为0。

注意我们定义的amount和currency方法，都没有括号或参数列表（甚至连空的都没有）。这是告诉编译器，在调用这个类的amount和currency方法时不需要括号，像这样：

```
val wages = new CashFlow(2000.0)
println(wages.amount)
println(wages.currency)
```

Scala对类的定义基本都能对应到Java中。但在面向对象的继承方式上，Scala所采用的方式跟Java有显著的差异。下一节就来讨论它们的差异。

9.4.3 特质

特质是Scala面向对象编程方式的主要组成部分。广义上来说，它们和Java接口一样。但跟Java接口不同的是，特质中可以给出方法的实现，并且这些实现可以由具备该特质的不同类共享。

要理解它所解决的Java问题，请看图9-2中从不同的基础类继承而来的两个Java类。如果这两

个类都要具备额外的相同功能，Java中的做法是声明它们实现了相同的接口。

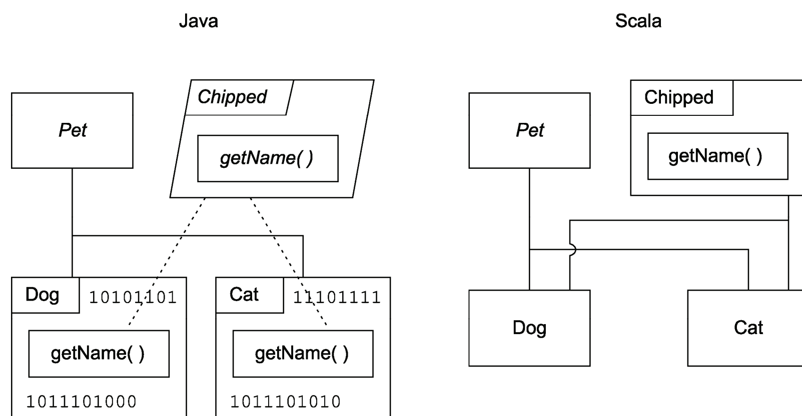


图9-2 Java模型中的实现复制

代码清单9-2是一个简单的Java例子，就是上面这种情况的代码。回忆一下4.3.6节那个兽医诊所的例子。很多带到诊所的动物都会被植入芯片，以便于识别。比如猫和狗几乎肯定会这么处理，但其他物种可能不会。

植入芯片的功能需要提取到单独的接口中。我们来修改一下代码清单4-11中的Java代码，加入这一功能（为了让代码看起来更清晰，我们省略了examine()方法）。

代码清单9-2 说明实现代码的复制

```
public abstract class Pet {
  protected final String name;

  public Pet(String name_) {
    name = name_;
  }
}

public interface Chipped {
  String getName();
}

public class Cat extends Pet implements Chipped {
  public Cat(String name_) {
    super(name_);
  }

  public String getName() {
    return name;
  }
}

public class Dog extends Pet implements Chipped {
  public Dog(String name_) {
    super(name_);
```

```
    }

    public String getName() {
      return name;
    }
  }
```

Dog和Cat中都有同样的getName()代码，因为Java接口中不能有实现代码。代码清单9-3是Scala用特质实现的版本。

代码清单9-3　用Scala实现的宠物类

```
class Pet(name : String)

trait Chipped {
  var chipName : String
  def getName = chipName
}

class Cat(name : String) extends Pet(name : String) with Chipped {
  var chipName = name
}

class Dog(name : String) extends Pet(name : String) with Chipped {
  var chipName = name
}
```

Scala要求在子类中必须给父类构造方法中出现的参数赋值。但在特质中声明的方法都会被子类继承。这样就减少了重复实现。你看，Cat和Dog类都要给参数name赋值。两个子类都可以访问Chipped中的实现——在此例中，参数chipName可以用来保存写在芯片上的宠物的名字。

9.4.4　单例和伴生对象

我们来看看Scala中的单例对象（即用关键字object定义的类）是如何实现的。回想一下9.1.1中的HelloWorld：

```
object HelloWorld {
  def main(args : Array[String]) {
    val hello = "Hello World!"
    println(hello)

  }
}
```

如果这是Java，你会觉得这段代码应该变成一个HelloWorld.class文件。实际上，Scala会把它编译成两个文件：HelloWorld.class和HelloWorld$.class。

因为这就是普通的类文件，所以你可以用第5章介绍的反编译工具javap看看Scala编译器产生的字节码。这会让你对Scala的类型模型及其实现方式有更多的了解。代码清单9-4是对这两个文件运行javap -c -p产生的结果：

代码清单9-4 反编译Scala的单例对象

```
Compiled from "HelloWorld.scala"
public final class HelloWorld extends java.lang.Object {
  public static final void main(java.lang.String[]);
    Code:
       0: getstatic     #11
➥ // Field HelloWorld$.MODULE$:LHelloWorld$;        <-- 取得单例伴生模块
       3: aload_0
       4: invokevirtual #13
➥ // Method HelloWorld$.main:([Ljava/lang/String;)V      <-- 调用伴生的main()方法
       7: return
}

Compiled from "HelloWorld.scala"
public final class HelloWorld$ extends java.lang.Object
➥ implements scala.ScalaObject {
  public static final HelloWorld$ MODULE$;        <-- 单例伴生实例

  public static {};
    Code:
       0: new           #9   // class HelloWorld$
       3: invokespecial #12  // Method "<init>":()V
       6: return

  public void main(java.lang.String[]);
    Code:
       0: getstatic     #19  // Field scala/Predef$.MODULE$:Lscala/Predef$;
       3: ldc           #22  // String Hello World!
       5: invokevirtual #26
➥ // Method scala/Predef$.println:(Ljava/lang/Object;)V
       8: return
                                                   私有构造方法
  private HelloWorld$();        <--
    Code:
       0: aload_0
       1: invokespecial #33  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: putstatic     #35  // Field MODULE$:LHelloWorld$;
       8: return
}
```

明白“Scala没有静态方法或域”这话是从何而来的了吗？除了这些结构，Scala编译器还自动生成了单例模式代码（不可变静态实例和私有构造方法），并把它们插到以`$`结尾的类中。`main()`方法仍然是常规的实例方法，但是是在单例的`HelloWorld$`类实例上调用的。

这意味着在这一对.class文件之间有二元性：一个和Scala文件的名字相同，另外一个加了个`$`。静态方法和域被放在了第二个单例类中。

Scala中名字相同的`class`和`object`非常常见。在这种情况下，单例类被当做了伴生对象。Scala源文件和两个VM类（主类和伴生对象）之间的关系如图9-3所示。

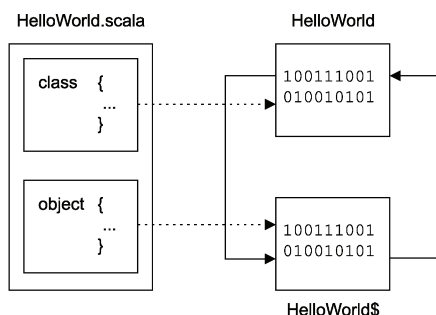


图9-3　Scala单例对象

尽管你不知道，但你确实已经遇到过伴生对象了。在HelloWorld中，你没必要指定println()方法在哪个类中。它看起来像个静态方法，所以你应该能想到它是伴生对象中的方法。

让我们再看一下代码清单9-2中与main()方法对应的字节码：

```
public void main(java.lang.String[]);
  Code:
     0: getstatic     #19  // Field scala/Predef$.MODULE$:Lscala/Predef$;
     3: ldc           #22  // String Hello World!
     5: invokevirtual #26
➥ // Method scala/Predef$.println:(Ljava/lang/Object;)V
     8: return
```

这段代码中的println()及其他随时可用的Scala函数都在Scala.Predef类的伴生对象中。

伴生对象在其相关类那里有特权。它能访问该类的私有方法。这使得Scala能以合理的方式定义私有辅助构造方法。Scala定义私有构造方法的语法是在其参数列表之前加上关键字private，像这样：

```
class CashFlow private (amt : Double, curr : String) {
  ...
}
```

如果私有的构造方法是主方法，那就只有两种办法可以创建该类的实例：或者通过伴生对象里的工厂方法（可以访问私有构造方法），或者调用一个公开的辅助构造方法。

接下来我们要进入下一主题：Scala的case类。你已经遇到过了，但为了刷新一下你的记忆，我们再重复一次，它们是通过自动提供一些基本方法来减少套路化代码的有效办法。

9.4.5　case类和match表达式

我们用Java实现一个简单的实体，比如Point类，如代码清单9-5所示。

代码清单9-5　一个用Java实现的简单类

```
public class Point {
  private final int x;
  private final int y;
```

```
  public Point(int x, int y) {          <-┐
    this.x = x;                           │
    this.y = y;                           │ 套路化代码
  }                                       │
                                          │
  public String toString() {          <-┘
    return "Point(x: " + x + ", y: " + y + ")";
  }
  @Override
  public boolean equals(Object obj) {   <-┐
    if (!(obj instanceof Point)) {        │
      return false;                       │
    }                                     │
    Point other = (Point)obj;             │ 套路化代码
    return other.x == x && other.y == y;  │
  }                                       │
                                          │
  @Override                               │
  public int hashCode() {               <-┘
    return x * 17 + y;
  }
}
```

这套路化代码简直太多了，而且更糟的是，像`hashCode()`、`toString()`、`equals()`以及所有的获取方法通常都是由IDE自动生成的。如果在语言内核的内部完成这些自动生成的工作，用更简单的语法岂不是更好？

Scala的确支持自动生成，`case`类就可以。代码清单9-5可以非常简单：

```
case class Point(x : Int, y : Int)
```

这和Java那段长长的代码功能一样，但除了更短，它还有别的好处。

比如说，用Java那个版本，如果要修改代码（假设要加个z坐标），就必须更新`toString()`和其他方法。实际上，应该要把原来那些方法全部删掉，然后让IDE再重新生成一次。

用Scala这些都没必要，因为根本就没显式定义需要跟着更新的方法。这归结为一个非常强的理论：不可能在没出现的源码中弄出bug来。

在创建新的`case`类实例时，关键字`new`可以省略。代码可以写成这样：

```
val pythag = Point(3, 4)
```

这样看来`case`类更像带一个或多个参数的枚举类型了。实际上`case`类的底层实现机制是提供一个创建新实例的工厂方法。

我们来看一下`case`类的主要用途：模式和`match`表达式。`case`类可以用在叫做构造器（Constructor）模式的Scala模式类型里，请看代码清单9-6。

代码清单9-6　`match`表达式中的Constructor模式

```
val xaxis = Point(2, 0)
val yaxis = Point(0, 3)
val some  = Point(5, 12)
```

```
val whereami = (p : Point) => p match {
  case Point(x, 0) => "On the x-axis"
  case Point(0, y) => "On the y-axis"
  case _           => "Out in the plane"
}
println(whereami(xaxis))
println(whereami(yaxis))
println(whereami(some))
```

我们在9.6节讨论actor和Scala的并发观点时会再次拜访Constructor模式和case类。

在结束本节之前，我们要发出一个警告。Scala丰富的语法和聪明的解析器能够用一些非常精炼和优雅的办法来表示复杂的代码。但Scala没有正式的语言规范，并且新特性的增加非常频繁。你应该多加小心——即便是经验丰富的Scala码农有时也会被语言特性出其不意的表现吓到。在语法特性互相结合时尤其如此。

我们来看一个例子：一种在Scala中模拟操作符重载的办法。

9.4.6 警世寓言

我们再想一想刚刚提到的Point case类。你可能想要用一种简单的办法来表示坐标的相加，或者坐标的线性增长。如果你数学好，可能马上就会意识到这是一个平面坐标上的向量空间属性。

代码清单9-7将方法定义得像普通的操作符一样。

代码清单9-7 模拟操作符重载

```
case class Point(x : Int, y : Int) {
  def *(m : Int) = Point(this.x * m, this.y * m)
  def +(other : Point) = Point(this.x + other.x, this.y + other.y)
}

var poin = Point(2, 3)
var poin2 = Point(5, 7)
println(poin)
println(poin 2)
println(poin * 2)
println(poin + poin2)
```

运行这段代码得到的输出应该是：

```
Point(2,3)
Point(5,7)
Point(4,6)
Point(7,10)
```

这下应该能看出Scala的case类跟Java里的等价物相比有多好了吧。只需要很少的代码，就能创造出一个很友好的类，产生合理的输出。定义+和*方法后，你已经可以模拟操作符重载了。

但这种方式有问题。请看下面这段代码：

```
var poin = Point(2, 3)
println(2 * poin)
```

这会导致编译错误：

```
error: overloaded method value * with alternatives:
  (Double)Double <and>
  (Float)Float <and>
  (Long)Long <and>
  (Int)Int <and>
  (Char)Int <and>
  (Short)Int <and>
  (Byte)Int
cannot be applied to (Point)
               println(2 * poin)
                         ^
one error found
```

尽管在case类Point上已经定义了方法*(m : Int)，但不是Scala要找的那个方法，所以出错了。为了让前面的代码编译成功，需要在Int类上实现*(p : Point)方法。这是不可能的，所以操作符重载只是一个假象。

这带出了Scala中有一个有趣的问题：很多语法特性的限制在某些情况下可能会让人大吃一惊。Scala的语言分析器和运行时环境在底层做了大量工作，但这些隐藏的机制是建立在尽量做正确的事的基础上的。

我们对Scala面向对象实现方式的介绍到这里就结束了。还有很多先进特性没涉及。很多现代化的类型系统和对象思想在Scala中都有实现，所以如果感兴趣，Scala的广阔天地对你来说大有可为。如果前面的那些内容勾起了你对Scala的类型系统和面向对象实现方式的兴趣，你可以去读一读Joshua Suereth的*Scala in Depth*（Manning，2012），或其他专门介绍Scala的图书。

你可能已经想到了，这些语言理论应用的一个重点是Scala的数据结构和集合，这也是我们下一节的主要内容。

9.5　数据结构和集合

你已经见过一个简单的Scala数据结构List了。它在任何编程语言中都是一个基本的数据结构，在Scala中也不例外。我们会花点时间探究一下List的细节，然后去研究一下Map。

接着我们会认真研究一下Scala中的泛型，包括与Java泛型的差别及Java泛型所不具备的能力。我们会以一些标准的Scala集合为例展开讨论，以便让你了解其原理。

先从Scala集合的几个一般性原则开始，特别是跟它的不可变性和它与Java集合的交互性相关的原则。

9.5.1　List

Scala中集合的实现方式跟Java很不一样。你可能会有点吃惊，因为在很多其他领域，Scala都在重用和扩展Java的组件、概念。

我们来看看Scala的理念所带来的最大差异：

- Scala集合通常都是不可变的；
- Scala把跟列表类似的集合的方方面面分解成了不同的概念；

- ❑ Scala构建`List`核心所涉及的概念非常少；
- ❑ Scala集合的实现方式是不同类型的集合提供的用户体验是一致的；
- ❑ Scala鼓励开发人员构建自己的集合类，并让它们用起来像内置的集合类一样。

我们会逐一讨论这些差异。

1. 不可变和可变集合

你首先要知道，Scala的集合既有不可变的版本，也有可变的版本，并且不可变版本是默认的（所有Scala源文件都可以随时访问）。

我们需要分辨可变集合和可变内容之间的本质区别。请看代码清单9-8。

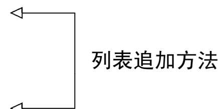
代码清单9-8　可变和不可变

```
import scala.collection.mutable.LinkedList
import scala.collection.JavaConversions._
import java.util.ArrayList

object ListExamples {
  def main(args : Array[String]) {
    var list = List(1,2,3)
    list = list :+ 4
    println(list)

    val linklist = LinkedList(1,2,3)
    linklist.append(LinkedList(4))
    println(linklist)

    val jlist = new ArrayList[String]()
    jlist.add("foo")
    val slist = jlist.toList
    println(slist)
  }
}
```



如上所示，`list`的引用是可变的（是`var`）。它指向一个不可变列表实例，所以可以通过重新赋值指向新对象。`:+`方法返回一个新的（不可变）`List`实例，这个新实例中含有新追加的元素。

相反，`linklist`是指向一个`LinkedList`的不可变引用（是`val`），而`LinkedList`实例是不可变的。`linklist`的内容可以修改，比如在其上调用`append()`。这种区别如图9-4所示。

代码清单9-8中还演示了一组转换函数：用来对Java集合和相应的Scala集合进行相互转换的`JavaConversions`类。

2. List的特质

Scala选择强调集合的特质和行为，这是它与众不同的另一个重要之处。我们以Java的`ArrayList`为例。除了`Object`，这个类还直接或间接地扩展了：

- ❑ `java.util.AbstractList`；
- ❑ `java.util.AbstractCollection`。

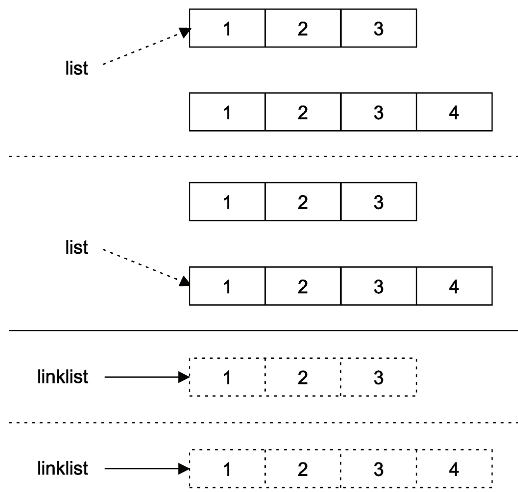


图9-4 不可变和可变集合

还有接口，ArrayList或它的某个父类实现了表9-2中列出的接口。

表9-2 ArrayList实现的Java接口

Serializable	Cloneable	Iterable
Collection	List	RandomAccess

对于Scala，情况要稍微复杂一点。以LinkedList为例，与它提供的功能相关的类或特质多达27个，如表9-3所示。

表9-3 **LinkedList**实现的Scala接口

Serializable	LinkedListLik	LinearSeq
LinearSeqLike	Cloneable	Seq
SeqLike	GenSeq	GenSeqLike
PartialFunction	Function1	Iterable
IterableLike	Equals	GenIterable
GenIterableLike	Mutable	Traversable
GenTraversable	GenTraversableTemplate	TraversableLike
GenTraversableLike	Parallelizable	TraversableOnce

Scala的集合类彼此之间的差异并不像Java那么明显。在Java中，List、Map、Set等，根据使用时的具体类型会有不同的处理模式。但在Scala中，由于使用了特质，类型的细化程度要比Java高得多。因此你可以把注意力放在集合的各种性质上，使用更加贴近需求的类型精确表达你的意图。

因此，Scala的集合处理代码要比Java的看起来更加整齐。

Scala中的set

如你所料，Scala既支持不可变的set，也支持可变set。set的典型用法跟Java里的模式一样：用一个中间对象按顺序遍历集合中的元素。但Java用的是Iterator或Iterable，而Scala用Traversable，它跟Java类型之间不能互操作。

构建列表的两个基础是：Nil表示空列表，::操作符能从已有的列表构建新列表。::操作符的发音是cons，它和Clojure的(concat)函数（见第10章）还有关系。这两者都表明Scala植根于函数式编程——最终可以追溯到Lisp中。

cons操作符有两个参数：一个类型为T的元素和一个类型为List[T]的对象。它会把两个参数合到一起创建一个新的List[T]值：

```
scala> val x = 2 :: 3 :: Nil
x: List[Int] = List(2, 3)
```

另外，也可以直接这样写：

```
scala> val x = List(2, 3)
x: List[Int] = List(2, 3)

scala> 1 :: x
res0: List[Int] = List(1, 2, 3)
```

cons操作符和括号

按cons操作符的定义，A :: B :: C的含义是没有歧义的，它的意思是A :: (B :: C)。这是因为::的第一个参数是单个类型为T的值。但A :: B是类型为List[T]的值，所以(A :: B) :: C作为可能的值没有任何意义。学院派的计算机科学家会说::是右相关性的。

这也解释了为什么要写成2 :: 3 :: Nil，而2 :: 3不行。::的第二个参数需要是List类型的值，而3不是List。

9.5.2　Map

映射也是一种经典的数据结构。Java最常见的就是它的HashMap。在Scala中，不可变的Map类是默认形态，而HashMap是标准的可变形态。

代码清单9-9中有几种简单、标准的映射定义和操作。

代码清单9-9　Scala中的Map

```
import scala.collection.mutable.HashMap

var x = Map(1 -> "hi", 2 -> "There")
for ((key, vau) <- x) println(key + ": " + vau)
x = x + (3 -> "bye")

val hm = HashMap(1 -> "hi", 2 -> "There")
hm += (3 -> "bye")
println(hm)
```

看到了吧，Scala定义映射字面值的语法简洁可爱：`Map(1 ->"hi", 2 -> "There")`。用箭头符号直观地表明了每个键“指向”的值。要从映射中取回值，请用`get()`方法，跟Java一样。

可变和不可变映射都用`+`表示向映射中添加元素（`-`表示移除）。但这个有些微妙，当用在可变映射上时，`+`修改映射然后返回它。而用在不可变实例上时，返回的是一个包含新的键/值对的新映射。这会导致`+=`操作符出现以下边界情况：

```
scala> val m = Map(1 -> "hi", 2 -> "There", 3 -> "bye", 4 -> "quux")
m: scala.collection.immutable.Map[Int,java.lang.String]
➥ = Map(1 -> hi, 2 -> There, 3 -> bye, 4 -> quux)

scala> m += (5 -> "Blah")
<console>:10: error: reassignment to val
       m += (5 -> "Blah")
         ^
scala>  val hm = HashMap(1 -> "hi", 2 -> "There", 3 -> "bye", 4 -> "quux")
hm: scala.collection.mutable.HashMap[Int,java.lang.String]
➥ = Map(3 -> bye, 4 -> quux, 1 -> hi, 2 -> There)

scala> hm += (5 -> "blah")
res6: hm.type = Map(5 -> blah, 3 -> bye, 4 -> quux, 1 -> hi, 2 -> There)
```

这是因为`+=`在不可变和可变映射中的实现是不一样的。对于可变映射，`+=`是一个方便修改映射的方法。这就是说在一个`val`映射上调用这个方法完全合法（就像Java在`final HashMap`上调用`put()`一样）。对于不可变映射，`+=`被分解成`=`和`+`的组合，就像在代码清单9-9里一样。它不能用在`val`上，因为`val`不允许再次赋值。

代码清单9-9中还有一个不错的语法：`for`循环。这用到了列表推导式（见9.3.5节）的思想，但结合了把键值对拆分成键和值的做法。这称为对解构，是Scala中另一个继承自函数式编程的概念。

对于Scala中的映射和它们的能力，我们仅仅触及了冰山一角，但我们要前往下一个主题了：泛型。

9

9.5.3 泛型

你已经知道了，Scala用方括号表示参数化类型，而且你也已经见过一些基本的Scala数据结构了。我们继续深入，看看Scala对泛型的处理方式跟Java有什么不同。

首先，如果在定义函数的参数类型时忽略掉了泛型，看看会发生什么：

```
scala> def junk(x : List) = println("hi")
<console>:5: error: type List takes type parameters
       def junk(x : List) = println("hi")
                    ^
```

在Java中，这是完全合法的。编译器可能会抱怨，但不会报错。而在Scala中，这是一个编译时错误。列表（和其他泛型）必须参数化——故事讲完了，Scala没有Java“生类型”的概念。

1. 泛型的类型推断

把泛型赋值给一个变量时，Scala会对类型参数做出恰当的类型推断。这符合Scala一贯坚持的类型推断和尽可能去掉套路化代码的风格：

```
scala> val x = List(1, 2, 3)
x: List[Int] = List(1, 2, 3)
```

Scala泛型中有个特性乍一看可能觉得奇怪，我们用:::操作符演示一下，看到下面两个列表联接起来产生了新的列表，你就明白为什么说它奇怪了：

```
scala> val y = List("cat", "dog", "bird")
y: List[java.lang.String] = List(cat, dog, bird)
scala> x ::: y
res0: List[Any] = List(1, 2, 3, cat, dog, bird)
```

奇怪吧，这样居然都不报错，还产生了新的`List`。运行时产生了一个`Int`和`String`的最小公父类（`Any`）的列表。

2. 泛型示例：候诊的宠物

假设有些宠物在等着看兽医，而你要建立候诊室里排队队列的模型。代码清单9-10是个不错的起点，用的是一些你已经熟悉的基础类和辅助函数。

代码清单9-10 候诊的宠物

```
class Pet(name : String)
class Cat(name : String) extends Pet(name : String)
class Dog(name : String) extends Pet(name : String)
class BengalKitten(name : String) extends Cat(name : String)

class Queue[T](elts : T*) {
  var elems = List[T](elts : _* )          <-- 需要类型提示

  def enqueue(elem : T) = elems ::: List(elem)

  def dequeue = {
    val result = elems.head
    elems = elems.tail
    result
  }
}

def examine(q : Queue[Cat]) {
  println("Examining: " + q.dequeue)
}
```

我们来考虑一下在Scala提示符中怎么使用这些类。这些是最简单的例子：

```
scala> examine(new Queue(new Cat("tiddles")))
Examining: line5$object$$iw$$iw$Cat@fb0d6fe

scala> examine(new Queue(new Pet("george")))
<console>:10: error: type mismatch;
 found   : Pet
 required: Cat
       examine(new Queue(new Pet("george")))
                         ^
```

到目前为止都很像Java。我们再多做几个简单的例子：

```
scala> examine(new Queue(new BengalKitten("michael")))
Examining: line7$object$$iw$$iw$BengalKitten@464a149a

scala> var kitties = new Queue(new BengalKitten("michael"))
kitties: Queue[BengalKitten] = Queue@2976c6e4
```

```
scala> examine(kitties)
<console>:12: error: type mismatch;
 found   : Queue[BengalKitten]
 required: Queue[Cat]
 examine(kitties)
         ^
```

这也相当平常。第一个例子没有将kitties作为临时变量，Scala的类型推断把队列的类型作为Queue[Cat]，并接受了michael的加入，因为它的类型是Cat的子类BengalKitten。第二个例子中，创建了变量kitties，显式声明了其类型。也就是说Scala不能用类型推断，所以不能接受类型不匹配的参数。

接下来我们去看看如何用类型系统的*类型变体*解决这些类型问题，特别是*协变*（类型变体还有其他形态，但协变最常用）。在Java中，这非常灵活，但也有点神秘。Scala和Java的做法我们都会演示一下。

3. 协变

“在Java中，List<String>是List<Object>的子类吗？”如果你问过类似问题，那这个话题就是为你准备的。

默认情况下，Java对这个问题的回答是“不是”，但你可以让它变成“是”。要知道怎么做，请看下面的代码：

```
public class MyList<T> {
  private List<T> theList;
}

MyList<Cat> katzchen = new MyList<Cat>();
MyList<? extends Pet> petExt = petl;
```

? extends Pet从句表示petExt是一个部分未知的类型参数（Java类型中的?读作“未知”）。可以确定的是MyList的类型参数必须是Pet或Pet的子类。这样在将类型参数为其子类的值赋给petExt时，Java编译器就不会阻拦。

这就相当于把MyList<Cat>变成了MyList<? extends Pet>的子类。注意，这种子类关系是在使用MyList类型时建立起来的，而不是定义时。类型的这个特性称为*协变*。

Scala的做法跟Java不同。它不是在使用类型时定义类型变体，而是在类型声明时显式指定协变。这样做有几个优势：

- 编译器可以在编译时检查不符合协变的使用；
- 所有概念上的思虑都交给了类型编写者，而不是抛给类型的使用者；
- 这样可以在基础集合类型间植入直观的关系。

理论上来说，这样的确不如Java那样使用现场的变体更灵活，但在实际应用中，Scala采取的方式所带来的好处完全可以抵消这种不便。大多数程序员很少会使用Java泛型中那些真正先进的特性。

Scala的标准集合，比如List，都实现了协变。这就是说List[BengalKitten]是List[Cat]的子类，而它又是List[Pet]的子类。我们来实际操练一下，请启动解释器：

```
scala> val kits = new BengalKitten("michael") :: Nil
kits: List[BengalKitten] = List(BengalKitten@71ed5401)

scala> var katzen : List[Cat] = kits
katzen: List[Cat] = List(BengalKitten@71ed5401)

scala> var haustieren : List[Pet] = katzen
haustieren: List[Pet] = List(BengalKitten@71ed5401)
```

我们在`var`上显式声明了类型，以免Scala把类型推断得过窄。

对Scala泛型的简略探讨到这里就结束了。下一个大主题是Scala在并发实现方式上的创新：放弃了多线程显式管理的方式，而选用了actor模型。

9.6　actor 介绍

Java的显式锁和同步模型刻下了岁月的痕迹。在最初设计Java语言时，它是一个奇妙的创新，但也埋下了祸根。Java并发模型本质上是面对两难境地时采取折中策略的产物。

锁太少，会导致并发代码不安全，出现竞态条件。锁太多，系统会丧失活力，代码瘫痪，工作毫无进展。这就是我们在第4章讨论过的，安全性与系统活力之间的矛盾。

使用基于锁的模型，必须照顾到给定时间内所有可能发生的并发操作。但随着程序变得越来越大，要做到滴水不漏会变得越来越困难。尽管Java有办法缓解一些问题，但核心问题还在，如果Java语言不能发布一个拒绝向后兼容的版本，就不可能从根本上解决这个问题。

非Java语言有机会从头开始。备选语言可以不向程序员暴露锁和线程的底层细节，而是在自己的运行时环境中提供额外的并发支持。

这应该没什么好奇怪的。毕竟在Java刚刚出现时，Java内存模型就受到过质疑。当时很多C和C++开发人员都对这种想法感到诧异，怎么能由运行时负责管理内存，而让开发人员远离这些细节呢?

我们来看一下Scala基于actor技术的并发模型，看它如何让并发编程变了样（也更简单）。

9.6.1　代码大舞台

actor是扩展`scala.actors.Actor`，并实现了`act()`方法的对象。希望这个定义能跟你脑海中对Java线程的定义相呼应。它们最大的差别就是actor在大多数情况下都不会通过共享的数据进行沟通。

程序员在共享数据时必须采用最佳实践。如果你想在actor间共享状态，Scala不会阻止你。我们只是认为这么做不好。actor有沟通的渠道：mailbox，从另一个上下文中发送过来的消息（工作项）可以放在mailbox中交给actor，请参见图9-5。

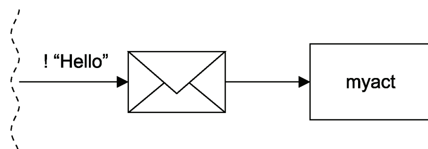


图9-5 scala的actor和mailbox

要创建actor，扩展Actor类就行：

```
import scala.actors._

class MyActor extends Actor {
  def act() {
    ...
  }
}
```

这看起来跟Java代码中声明Thread的子类很像。跟线程一样，我们也要告诉actor开始启动，并进入消息接收的状态，这要调用start()方法。

Scala同样提供了创建actor的工厂方法actor（与Java里创建Runnable匿名实现类的静态工厂方法相对应）。用它写出来的Scala代码很精炼：

```
val myactor = actor {
      ...
}
```

传给actor的代码块会变成act()方法中的内容。另外，这样创建的actor不需要再单独调用start()，它会自动启动。

这是一块香甜的语法糖，但我们还要介绍Scala并发模型的核心部件mailbox，所以别回味了，现在就去看看吧。

9.6.2 用mailbox跟actor通信

从另一个对象给actor发消息很简单，只要在actor对象上调用!方法就行了。

然而在接收端要有代码处理这些消息，否则它们就会堆在mailbox里。另外，actor方法体通常需要有个循环，以便能处理所有流入的消息。我们在Scala REPL中实际操练一下：

```
scala> import scala.actors.Actor._
       val myact = actor {
         while (true) {
           receive {
             case incoming => println("I got mail: "+ incoming)
           }
         }
       }
myact: scala.actors.Actor = scala.actors.Actor$$anon$1@a760bb0

scala> myact ! "Hello!"
I got mail: Hello!

scala> myact ! "Goodbye!"
I got mail: Goodbye!

scala> myact ! 34
I got mail: 34
```

上面代码中的receive方法就是actor对消息的处理。而工厂方法的参数（代码块）则是消息处理方法的主体。

注意　总体来说，Scala模型跟我们第4章（代码清单4-13）讨论的处理模式相似，Java处理线程相当于acctor的角色，`LinkedBlockingQueue`相当于Scala中的mailbox。Scala只是以非常直白的方式为这种模式提供了语言和类库层面的支持，可以大量减少使用这种模式时所要编写的套路化代码。

尽管这个例子非常简单，但也包含了很多使用actor的基础知识：

- 在actor方法中要用循环的方式处理接收消息流；
- 用`receive`方法处理接收到的消息；
- 用一组`case`作为`receive`的主体。

最后这点还得继续讨论。这一组`case`被称为偏函数①。之所以要这样用，是因为Scala中的actor还有一点比Java方便。具体来说就是mailbox是不区分类型的。也就是说你可以向actor发送任何类型的消息，actor可以用类型化模式和构造器模式接收不同类型的消息。

除了这些基础知识，这里还有一些使用actor的最佳实践。编写代码应该尽量遵循下面几条规则：

- 把传入消息做成不可变的；
- 考虑把消息类型做成`case`类；
- 不要在actor内部做阻塞操作，一个也别做。

不是每一个程序都需要遵守所有的最佳实践，但大多数应用程序应该都能从这些建议中受益。

对于更加复杂的actor，经常有必要控制它的启动和关闭。关闭actor通常都是用带有`Boolean`条件判断的循环。如果你喜欢，也可以将actor写成函数式的风格，这样传入的消息就不会影响它的状态。

Scala对基于actor的并发编程提供的支持还有很多。我们在这里看到的只是皮毛。如果想全面了解，请参阅Nilanjan Raychaudhuri的大作*Scala in Action*（Manning, 2010）。

9.7　小结

Scala跟Java有显著的差异：

- 支持更灵活的函数式编程风格；
- 类型推断使静态语言用起来有动态语言的感觉；
- Scala先进的类型系统扩展了Java的面向对象概念。

下一章会介绍最后一门非Java语言：Lisp方言Clojure，这可能是从各方面来看都最不像Java的语言。我们会以不可变性、函数式编程和另一种并发为基础展开讨论，并展示Clojure如何利用这些思想构建起了一个强大无比、美丽异常的编程环境。

---

① 在Scala中，偏函数是指类型为`PartialFunction[-A,+B]`的函数。`A`是其接受的函数类型，`B`是其返回的结果类型。偏函数最大的特点就是它只接受其参数定义域的一个子集，而对于这个子集之外的参数则抛出运行时异常。这与`case`语句非常契合，因为我们在使用`case`语句时常常是匹配一组具体的模式，最后用“_”来代表剩余的模式。如果一组`case`语句没有涵盖所有的情况，那么这组`case`语句就可以被看做是一个偏函数。——译者注

第 10 章

Clojure：更安全地编程

本章内容

- ❑ Clojure实体和状态的概念
- ❑ Clojure的REPL
- ❑ Clojure语法、数据结构和序列
- ❑ Clojure与Java的交互能力
- ❑ Clojure的多线程开发
- ❑ 软件事务内存

Clojure跟Java以及我们前面研究的语言差别很大。Clojure是在JVM上重新实现的Lisp。Lisp是最古老的编程语言，如果你对它还不熟悉，没关系。与Lisp语言家族有关的一切，只要是你需要知道的，我们都会告诉你，你可以安心开始Clojure之旅。

除了从Lisp继承的强大编程技术，Clojure还增添了一些令人惊叹的前沿技术。这种组合让Clojure从JVM语言中脱颖而出，成为应用程序开发的诱人选择。

Clojure中的并发工具包和数据结构就是一项新技术。并发抽象层让程序员可以写出更加安全的多线程代码。它和Clojure的序列抽象层（对集合和数据结构上的不同看法）相结合，为开发人员提供了非常强大的工具箱。

想掌握这些力量，先要了解Clojure跟Java在编程方式上截然不同的理念。这种差异使得Clojure学起来很有趣，并且很可能会改变你的思考方式。不管你用的是什么语言，学习Clojure都会让你成为更好的程序员。

我们一开始会先讨论Clojure处理状态和变量的方式。在给出一些简单的例子后，会介绍这门语言的基本词汇表——用来构建语言其余部分的特殊形态。我们将深入到Clojure的语法中，了解它的数据结构、循环和函数。然后介绍序列，这是Clojure最强的抽象概念之一。我们会用两个非常引人注目的特性来给这一章收尾：跟Java的紧密集成以及Clojure惊人的并发支持。

10.1 Clojure 介绍

我们先来看Clojure跟Java在理念上最重要的差别，即对状态，变量和存储的不同认识。如图

10-1所示，Java（跟Groovy和Scala一样）有一个内存和状态模型，把变量当作保存可变内容的“盒子”（内存位置）。

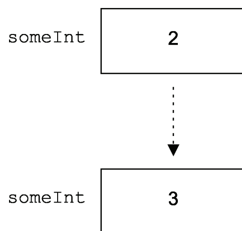


图10-1　命令式语言的内存使用

而Clojure认为值才是真正重要的概念。值可以是数字、字符串、向量、映射、集合，或其他任何东西。一旦创建，值就再也不会改变。这一点真的非常重要，所以我们要再重复一次。一旦创建，Clojure的值就不能再变了，因为它们是不可变的。

这就是说命令式语言那种装着可变内容的盒子模型不是Clojure思考问题的方式。图10-2是Clojure处理状态和内存的方式。它在名字和值之间创建了一个关联关系。

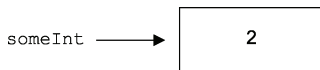


图10-2　Clojure的内存使用

这就是绑定，通过特殊形式`(def)`建立。Clojure中的特殊形式相当于Java的关键字，但请注意，Clojure中的术语“关键字”含义不同，稍后我们会介绍。

`(def)`的句法是：

```
(def<名称> <值>)
```

如果你觉得这个句法看起来有点怪异，不要担心，这完全是Lisp的普通句法，你很快就会习惯的。现在你可以假装是在调用下面这样一个方法，只是括号的位置不太一样：

```
def(<名称>, <值>)
```

接下来我们要在Clojure的交互式环境中写一个久经考验的例子，演示一下`(def)`的用法。

10.1.1　Clojure的Hello World

如果你还没装Clojure，请参见附录D。然后切换到Clojure所在的目录，运行如下命令：

```
java -cp clojure.jar clojure.main
```

这个命令会启动Clojure的REPL环境。在编写Clojure代码时，你会在这个交互环境里花上很多时间。

`user=>`是Clojure的会话提示符，你可以把这个会话环境当做高级的调试环境，或者命令行工具：

```
user=> (def hello (fn [] "Hello world"))
#'user/hello
user=> (hello)
"Hello world"
```

这段代码一开始先给标识符hello绑定一个值。(def)就是用来建立标识符（Clojure称为符号）和值之间的绑定关系的。底层实现的时候，它也会创建一个对象var，用来表示这种绑定关系（和符号的名字）。

那这里绑定的值是什么？这个值是：

```
(fn [] "Hello world")
```

这是一个函数，在Clojure中也是一个纯正的值（因此也是不可变的）。这个函数没有参数，返回字符串"Hello world"。

绑定之后，可以用(hello)执行。Clojure运行时会输出该函数的计算结果，也就是"Hello world"。

现在，应该录入这个例子（如果你还没做），看看它的表现是不是跟我们说的一样。完成之后，我们就可以继续探索了。

10.1.2　REPL入门

在REPL中可以输入Clojure代码，也可以执行Clojure函数。它是个交互式环境，而且在前面得出的计算结果不会被丢掉。可以用它做*探索式编程*，我们会在10.5.4节讨论这种编程方式，基本就是不断试验代码。用Clojure开发经常都是先在REPL里把代码调好，然后用正确的构件搭出越来越大的函数。

马上看一个例子。先声明，再次调用def可以改变符号和值的绑定关系，我们在REPL中看一下。代码中用的实际上是(def)的变体(defn)：

```
user=> (hello)
"Hello world"
user=> (defn hello [] "Goodnight Moon")
#'user/hello
user=> (hello)
"Goodnight Moon"
```

注意，hello最初的绑定关系一直都在，直到被你改掉，这是REPL的一个关键特性。这还是状态，只不过换了个说法，变成了哪个符号绑定到哪个值上，并且这个状态存在于用户输入的不同行间。

Clojure中没有可变状态，但有可以改变绑定值的符号。Clojure不是让“内存盒子”中的内容改变，而是让符号绑定到不同的不可变值上。换句话说就是在程序的生命期内，var可以指向不同的值。请参见图10-3。

注意　可变状态和不同绑定两者之间的区别很微妙，但这个概念很重要，一定要掌握。要记住，可变状态是指盒子中的内容变了，而重新绑定是指在不同时间指向不同的盒子。

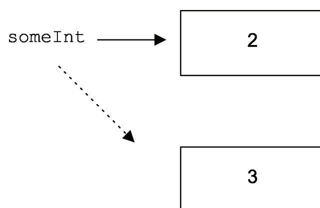


图10-3　可以改变的Clojure绑定

这段代码中还溜进了另一个Clojure概念，“定义函数”宏(defn)。宏是类Lisp语言的关键概念之一，其核心思想是内置结构和普通代码之间的区别应该尽可能小。

用宏可以创建跟内置语法类似的形式。创建宏是高级话题，但掌握了它之后，你就能制造出非常强大的工具。

这就是说语言真正的原语系统（特殊形式）可以用一种几乎无法察觉的方式构建起整个语言的核心。宏(defn)就是这种构建的产物。它只是将函数值绑定到符号的相对简单的方法（当然，要创建合适的var）。

10.1.3　犯了错误

如果你犯错了，会怎么样？比如你漏掉了[]（函数声明的一部分，表明这个函数没有参数）。

```
user=> (defn hello "Goodnight Moon")
#'user/hello
user=> (hello)
java.lang.IllegalArgumentException: Wrong number of args (0) passed to:
user$hello (NO_SOURCE_FILE:0)
```

所有后果只是hello标识符绑定到了一个未知的东西上。你可以在REPL中重新绑定来修复它：

```
user=> (defn hello [] (println "Dydh da an Nor"))
➥ ; "Hello World" in Cornish
#'user/hello
user=> (hello)
Dydh da an Nor
nil
user=>
```

跟你猜的一样，上面这段代码中的分号（;）表示直到行尾的内容都是注释，(println)是输出字符串的函数。注意看(println)，它跟所有函数一样，返回了一个值，在函数执行结束后回显到REPL中。结果值是nil，相当于Java里的null。

10.1.4　学着去爱括号

奇思妙想和幽默感是程序员文化不可或缺的一部分。说Lisp是“很多烦人的傻括号”的缩写就是个很古老的笑话。其实Lisp是列表处理（List Processing）的缩写，真相就是这么平淡无奇。很多Lisp程序员都用这个笑话自嘲，因为它确实戳到了Lisp语法的痛处。

实际上，这个障碍被夸大了。Lisp句法的确特立独行，但也不像看起来那么碍手碍脚。另外，Clojure还为减轻入门的障碍做了几项创新。

我们再看一下Hello World。调用返回“Hello World”的函数写成：

```
(hello)
```

用Java写应该是这样（假设你已经在某个类里定义了hello方法）：

```
hello();
```

但Clojure的表达式不是myFunction(someObj)，而是(myFunction someObj)。这种写法叫波兰表示法，因为它是19世纪的波兰数学家发明的。

如果你研究过编译原理，可能想知道这是否和抽象语法树（AST）之类的概念有关。简单地说是“有”。可以证明，用波兰表示法（Lisp程序员通常管它叫s表达式）写成的Clojure或其他Lisp程序是其简单直接的AST表示。

你可以认为Lisp程序是直接用AST写的。Lisp程序的数据结构表示和代码没有本质上的差别，所以代码和数据是完全可以互换的。这也是Clojure的表示法看起来有点奇怪的原因——类Lisp语言用它来模糊内置的原生代码、用户代码和类库代码之间的区别。对于Java程序员来说，这股强大力量对他们的引力要远远超过稍微有点古怪的语法。

让我们更深入地学一些Clojure语法，然后用它写一些真正的程序。

10.2 寻找 Clojure：语法和语义

我们上一节介绍了(def)和(fn)两个特殊形式（special form）。这里还有几个需要你马上掌握的特殊形式，它们构成了语言的基础词汇表。Clojure中还有大量实用的形式和宏，用得越多，认识会越来越深刻。

Clojure中的函数非常多，托它们的福，你能想到的任务很多都可以用Clojure完成。不要因此而沮丧，你应该感到庆幸。你要干的活大部分都有人替你干了，不该高兴吗？

我们在这一节会讨论特殊形式的基本工作集，然后是Clojure的原生数据类型（相当于Java的集合）。之后会接着讨论Clojure代码的自然编写风格——以函数而不是变量为中心。JVM面向对象的性质在底层还会存在，但Clojure强调函数的那种力量在纯粹的面向对象语言中表现得不太明显。

10.2.1 特殊形式新手营

表10-1给出了一些最常用的Clojure特殊形式。你现在最好快速地把这张表过一遍，然后在10.3节遇到具体例子时再回来看看。

这个特殊形式列表不算详尽，并且其中很多特殊形式都有多种用法。表10-1中只是它们的基本用例，而且都不全面。

表10-1　Clojure一些基本的特殊形式

特殊形式	含　　义
(def<符号> <值?>)	把符号绑到值上（如果有的话）。如有必要创建与符号对应的var
(fn<名称>? [<参数>*]<表达式>*)	返回带有特定参数的函数值，并把它们应用到表达式上。通常跟(def)相结合，变成形式(defn)
(if<test> <then> <else>?)	如果test的计算结果为true，计算then并产出其结果。否则计算else并产出其结果，当然，前提是else存在
(let[<绑定>*] <表达式>*)	给局部名称分配别名值，并隐式定义一个作用域。使得在let作用域内的所有表达式都能获得该别名
(do<表达式>*)	按顺序计算表达式的值，并产出最后一个的结果
(quote<形式>)	照原样返回形式（不经计算）。它只能接受一个形式参数，其他的参数全都会被忽略
(var<符号>)	返回与符号对应的var（返回一个Clojure JVM对象，不是值）

现在你对一些特殊形式的基本语法有进一步的了解了，让我们转去看看Clojure的数据结构吧，也看看它们怎么操作数据。

10.2.2　列表、向量、映射和集

Clojure中有几个原生数据类型。用的最多的是列表（list），即单向链表。

列表通常都用括号围起来，因为形式一般也是用圆括号，所以这算是一个轻微的语法障碍。况且括号还用来调用函数。所以初学者经常会犯下面这种错误：

```
1:7 user=> (1 2 3)
java.lang.ClassCastException: java.lang.Integer cannot be cast to
clojure.lang.IFn (repl-1:7)
```

之所以会出错，是因为Clojure中的值非常灵活，它希望第一个参数是函数值（或绑定到函数值上的符号），把2和3当做这个函数的参数。可在上例中1不是函数值，所以Clojure无法编译。按我们的说法，这个s表达式是无效的。只有有效的s表达式才能作为Clojure形式。

解决办法是用(quote)形式，它的缩写是'。所以我们可以用两种方式定义列表：

```
1:22 user=> '(1 2 3)
(1 2 3)
1:23 user=> (quote (1 2 3))
(1 2 3)
```

(quote)以一种特殊的方式处理它的参数。具体来说就是它不会计算参数，所以第一个参数不是函数值也没问题。

Clojure的向量（vector）跟数组类似，实际上，基本上可以把Clojure列表等同于Java的LinkedList，向量等同于ArrayList。向量可以用方括号表示，所以下面这些定义都一样：

```
1:4 user=> (vector 1 2 3)
[1 2 3]
1:5 user=> (vec '(1 2 3))
[1 2 3]
1:6 user=> [1 2 3]
[1 2 3]
```

在前面声明Hello World和其他函数时，就是用向量来表示函数的参数。注意，(vec)形式以一个列表为参数，并用这个列表创建向量，而(vector)形式以多个独立符号为参数，并返回包含它们的向量。

函数(nth)有两个参数：集合和索引。它跟Java中List接口的get()方法类似。可以用在向量和列表上，也可以用在Java集合甚至字符串（字符的集合）上，请看下例：

```
1:7 user=> (nth '(1 2 3) 1)
2
```

Clojure也支持映射（map，相当于Java的HaspMap），定义很简单：

```
{key1 value1 key2 "value2}
```

从映射里取值也非常简单：

```
user=> (def foo {"aaa" "111" "bbb" "2222"})
#'user/foo
user=> foo
{"aaa" "111", "bbb" "2222"}
user=> (foo "aaa")
"111"
```

Clojure把前面带冒号的映射键称为“关键字”：

```
1:24 user=> (def martijn {:name "Martijn Verburg",
➥ :city "London", :area "Highbury"})
#'user/martijn
1:25 user=> (:name martijn)
"Martijn Verburg"
1:26 user=> (martijn :area)
"Highbury"
1:27 user=> :area
:area
1:28 user=> :foo
:foo
```

关于关键字，请记住下面这些知识点。

- Clojure的关键字是只有一个参数的函数，其参数必须是映射。
- 在映射上调用这个函数会返回映射里与该关键字函数对应的值。
- 关键字的使用遵循语法对称性规则，即(my-map :key)和(:key my-map)都是合法的。
- 关键字作为值使用时返回自身。
- 关键字在使用之前无需声明或def。
- Clojure中的函数也是值，因此可以放在映射里当键用。
- 可以用逗号（但没必要）来分隔键/值对，因为Clojure会把它们当做空格处理。
- 除了关键字，其他符号也能用在映射里做键，但关键字太好用了，所以我们要特别提出来，你应该把它用在自己的代码中。

除了映射字面值，Clojure还有个(map)函数。但不要上当，它不像(list)，(map)函数不会产生映射。而是对集合中的元素轮番应用其参数中的函数，并用返回的新值建立一个新集合（实际上是Clojure序列，请参见10.4节）。

```
1:27 user=> (def ben {:name "Ben Evans", :city "London", :area "Holloway"})
#'user/ben
1:28 user=> (def authors [ben martijn])
#'user/authors
1:29 user=> (map (fn [y] (:name y)) authors)
("Ben Evans" "Martijn Verburg")
```

(map)还有别的形式，可以一次处理多个集合，但一次输入一个集合的形式最常用。

Clojure也支持集（set），跟Java的HashSet很像。它的缩写形式是：

```
#{"apple" "pair" "peach"}
```

这些数据结构是构建Clojure程序的基础。

Java土著可能会感到吃惊，居然一直没有提到对象。这不是说Clojure不是面向对象的，但它对面向对象的观点的确和Java不一样。Java认为世界是由封装了数据和代码的静态数据类型组成的。而Clojure强调函数和形式，尽管这些在底层都是由JVM上的对象实现的。

Clojure和Java在世界观上的差别最终会体现在代码里。要充分理解Clojure的观点，必须用Clojure写些程序，并弄明白相比Java的面向对象结构它有哪些优势。

10.2.3　数学运算、相等和其他操作

Clojure没有Java里那种意义上的操作符。所以怎么才能，比如说，让两个数相加呢？在Java里这很容易：

```
3 + 4
```

但Clojure没有操作符，只能用函数：

```
(add 3 4)
```

这也挺好，但我们可以做得更好。因为Clojure里没有操作符，所以我们不用为它们保留任何字符。这就是说Clojure的函数名称可以更加稀奇古怪，所以我们可以这样写[①]：

```
(+ 3 4)
```

Clojure函数一般都支持变参（参数数量可变），比如还可以这样：

```
(+ 1 2 3)
```

这个运算结果是6。

Clojure的相等形式（相当于Java里的equals()和==）状况稍微有点复杂。Clojure有两个跟相等相关的形式：(=)和(identical?)。注意它们的名字，这全都是因为Clojure不用为操作符保留字符。另外，(=)也是等号，而不是赋值符号。

下面这段代码设置了一个列表list-int和一个向量vect-int，并比较它们是否相等：

① 例子中的(+)是clojure.core命名空间下的函数，能够接受0到任意数目的参数，假如没有参数，则返回0。所以虽然Clojure没有操作符，但有很多提供了操作符功能的核心函数，所以你大可不必担心怎么计算3 * 4，用早已准备好的函数(* 3 4)就行了。——译者注

```
1:1 user=> (def list-int '(1 2 3 4))
#'user/list-int
1:2 user=> (def vect-int (vec list-int))
#'user/vect-int
1:3 user=> (= vect-int list-int)
true
1:4 user=> (identical? vect-int list-int)
false
```

(=)形式会检查集合是否由相同的对象以相同的顺序组成的（list-int和vect-int符合这一要求），而(identical?)会检查它们是否真的是同一个对象。

你可能也注意到了，符号名称都没有用驼峰式大小写[①]。这在Clojure中很常见，符号通常都用小写，单词之间用连字符连接。

Clojure中的true与false

Clojure中有两个值表示逻辑假：false和nil。其他全是逻辑真。很多动态语言都这样，但对于Java程序员来说这有点奇怪。

掌握了基本的数据结构和操作符，让我们把之前见过的特殊形式和函数拼到一起，写一个稍微长点的Clojure函数吧。

10.3　使用函数和循环

从本节开始，我们会接触到Clojure中一些实质性的内容。从编写函数处理数据开始，让你看到Clojure对函数的重视程度。接着介绍循环结构，以及读取器（reader）宏和派发（dispatch）形式。最后，我们会以Clojure的函数式编程和闭包作为本节的收尾。

举例说明是好办法，所以我们先来几个简单的例子，然后朝Clojure提供的强大函数式编程技术进发。

10.3.1　一些简单的Clojure函数

代码清单10-1中定义了三个函数。其中两个是非常简单的单参函数，另一个稍微有点复杂。

代码清单10-1　定义简单的函数

```
(defn const-fun1 [y] 1)

(defn ident-fun [y] y)

(defn list-maker-fun [x f]
  (map (fn [z] (let [w z]
    (list w (f w))
 )) x))
```

① 驼峰式大小写（Camel-Case）一词来自Perl语言中普遍使用的大小写混合格式，而Larry Wall等人所著的畅销书 *Programming Perl: Unmatched power for text processing and scripting*（O'Reilly，2012）的封面图片正是一匹骆驼。——译者注

在这段代码中，(const-fun1)接受一个参数，返回1，(ident-fun)接受一个数值并返回数值本身。数学家会管它们叫*常量函数*和*恒等函数*。还有，函数定义中使用向量表示函数的参数，(let)形式中用的也是向量。

第三个函数比较复杂。函数(list-maker-fun)有两个参数：第一个是包含所处理的值的向量x，第二个一定是函数。

我们来看一下如何使用list-maker-fun，如代码清单10-2所示。

代码清单10-2　使用函数

```
user=> (list-maker-fun ["a"] const-fun1)
(("a" 1))
user=> (list-maker-fun ["a" "b"] const-fun1)
(("a" 1) ("b" 1))
user=> (list-maker-fun [2 1 3] ident-fun)
((2 2) (1 1) (3 3))
user=> (list-maker-fun [2 1 3] "a")
java.lang.ClassCastException: java.lang.String cannot be cast to
clojure.lang.IFn
```

把这些表达式敲到REPL中实际上是和Clojure的编译器交互。表达式(list-maker-fun [2 1 3] "a")之所以无法编译，是因为(list-maker-fun)的第二个参数应该是函数，而字符串显然不是。看到10.5节你就会知道，对于VM来说，Clojure函数是实现了clojure.lang.IFn的对象。

这个例子表明在跟REPL交互时仍然会涉及一些静态类型问题。因为Clojure不是解释型语言。即便是在REPL中，输入的每个Clojure形式都会被编译成JVM字节码并连接到运行时系统上。Clojure函数在定义完后就被编译成JVM字节码了，所以在出现静态类型冲突时VM会报出ClassCastException异常。

代码清单10-3中的Clojure代码更长。Schwartzian转换可有年头了，从20世纪90年代在Perl中出现后就一直在用。其基本思想是基于向量中元素的某些属性对元素进行排序。排序所依据的属性值是通过在元素上调用*键控函数*确定的。

代码清单10-3中定义的Schwartzian转换所调用的键控函数是key-fn。在真正调用(schwartz)函数时需要提供一个用作键控的函数。代码清单10-3中用的是我们的老朋友(ident-fun)。

代码清单10-3　Schwartzian转换

```
1:65 user=> (defn schwartz [x key-fn]
  (map (fn [y] (nth y 0))                  ◁── 第三步
       (sort-by (fn [t] (nth t 1))         ◁── 第二步
                (map (fn [z] (let [w z]    ◁── 第一步
                     (list w (key-fn w)))
                  )  x))))
#'user/schwartz
1:66 user=> (schwartz [2 3 1 5 4] ident-fun)
(1 2 3 4 5)
1:67 user=> (apply schwartz [[2 3 1 5 4] ident-fun])
(1 2 3 4 5)
```

这段代码分为三步：

- 创建一个包含键值对的列表；
- 基于键控函数的值对键值对排序；
- 仅从排好序的键值对列表中取出原始值，构建新列表（并抛弃键控函数值）。

如图10-4所示。

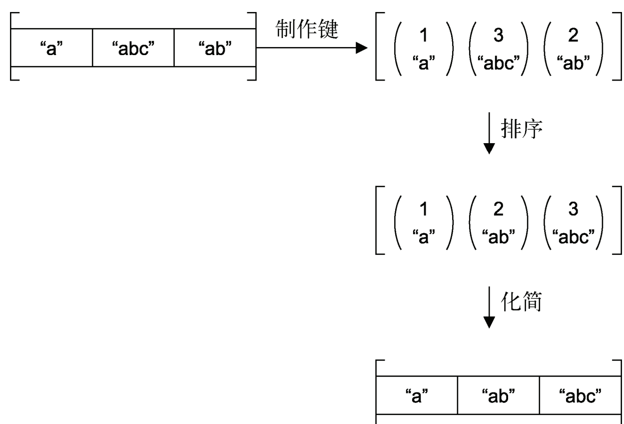


图10-4　Schwartzian转换

代码清单10-3中引入了一个新形式：`(sort-by)`。这个函数有两个参数：一个是用来排序的函数，一个是要排序的向量。还有`(apply)`形式，它也有两个参数：一个是要调用的函数，一个是传给它的向量参数。

Randall Schwartz最初用Perl编写Schwartzian转换（该转换以他的名字命名）时在刻意模仿Lisp。我们现在又用Clojure编写，算是绕了一圈又回来了。挺有意思！

Schwartzian转换的示例很实用，我们稍后还会用到它。因为它的复杂性足以用来阐明好几个概念。

接下来我们来讨论下Clojure的循环，可能和你所习惯的循环有点不太一样。

10.3.2　Clojure中的循环

Java里的循环相当简单直接，可选的循环有`for`、`while`，还有其他几种。其核心思想通常是重复一组指令，直到满足某一条件（一般用一个可变变量表示）。

这对Clojure是个小难题：举个例子，对于没有可变变量作为循环索引的Clojure，怎么表示`for`循环呢？在传统的Lisp中通常用递归形式实现循环。但JVM不能保证尾递归优化（Scheme和其他Lisp语言有这种要求），所以在Clojure中用递归可能会导致栈溢出。

而Clojure有不会增加栈空间占用的结构。最常用的是`loop-recur`，下面的代码展示了如何用`loop-recur`构建一个和`for`循环类似的结构。

```
(defn like-for [counter]
(loop [ctr counter]
  (println ctr)
  (if (< ctr 10)
      (recur (inc ctr))
      ctr
 )))
```

(loop)形式以包含符号局部名称的向量为参数——像(let)定义的别名。然后当执行到(recur)形式时（本例中只有ctr别名小于10才会执行该形式），它会将控制分支返回到(loop)形式中，但指定了新的值。这样我们就可以搭建循环式结构（比如for和while循环），但实现中仍有递归的味道。

现在我们转入下一主题，看一看Clojure语法的简写，帮你把程序写得更短、更精炼。

10.3.3　读取器宏和派发器

Clojure有些让很多Java程序员吃惊的语法特性。其中之一是没有操作符。它的副作用是放宽了Java对能用在名称中的字符的限制。你已经见过像(identical?)这样的函数了，这在Java中是非法的，但对于哪些字符不能用在符号中，我们还没有说明。

表10-2列出了不能用在Clojure符号中的字符。Clojure分析器保留了这些字符自用，它们通常被称为读取器宏。

表10-2　读取器宏

字　符	名　称	含　义
'	引号	展开为(quote)，产出不进行计算的形式
;	注释	标记直到行尾的注释，就像Java里的//
\	字符	产生一个字面字符
@	解引用	展开为(deref)，接受var对象并返回对象中的值（跟(var)形式的操作相反）。在事务内存上下文中还有其他含义（见10.6节）
^	元数据	将一个元数据的映射附加到对象上。请查阅Clojure文档了解详情
`	语法引用	经常用在宏定义中的引号形式，不太适合初学者。请查阅Clojure文档了解详情
#	派发	有几种不同的子形式，见表10-3

根据#后面的字符，派发读取器宏有几种不同的子形式，请见表10-3。

表10-3　派发读取器宏的子形式

派发形式	含　义
#'	展开为(var)
#{}	创建一个集字面值，在10.2.2节中用过
#()	创建匿名函数字面值，用在那些使用(fn)太啰嗦的地方
#_	跳过下一个形式。可以用#_(... 多行 ...)来创建多行注释
#"<模式>"	创建一个正则表达式（作为java.util.regex.Pattern对象）

关于派发形式，还有几点要提一下。变量引用形式#'解释了REPL执行(def)之后的表现：

```
1:49 user=> (def someSymbol)
#'user/someSymbol
```

(def)形式返回新创建的var对象，命名为someSymbol，驻留在当前的命名空间中（就是用户所在的REPL），所以#'user/someSymbol是(def)返回的完整值。

匿名函数字面值也是减少繁琐代码的创新。它省略了参数向量，用一种特殊的语法让Clojure读取器推断函数字面值需要多少个参数。

代码清单10-4是我们用这个语法重写的Schwartzian转换。

代码清单10-4　重写Schwartzian转换

```
(defn schwartz [x f]
  (map #(nth %1 0)
       (sort-by #(nth %1 1)
                (map #(let [w %1]
                     (list w (f w))
                    ) x))))
```

匿名函数字面值

用%1当做函数字面值参数的占位符（后续参数可以用%2、%3等）真的很好，这样的代码也更容易看懂。这种显而易见的线索对程序员很有帮助，就像你在9.3.6节见过的Scala函数字面值里的箭头符号一样。

Clojure严重依赖于以函数为基本计算单元的概念，而不像Java以对象为语言的根本。这种方式自然会导向函数式编程，也就是我们的下一主题。

10.3.4　函数式编程和闭包

我们现在要进入恐怖的Clojure函数式编程世界。或者，我们没有，因为它不恐怖。实际上，我们这一整章都在学习函数式编程，只是没告诉你，怕把你吓跑。

7.3.2节中说过，函数式编程意味着函数是一个值。函数可以传递，放在变量中操作，就像2或"hello"一样。但那又怎么样？我们回头看看第一个例子：(def hello (fn [] "Hello world"))。我们创建了一个函数（没有参数，返回字符串"Hello world"），把它绑定到符号hello上。函数仅仅是个值，本质上跟2这种值没什么区别。

在10.3.1节，我们以Schwartzian转换为例介绍了以另外一个函数为输入值的函数。这也不过是一个以特定类型为输入参数的函数，唯一的区别不过是这个类型是函数。

关于闭包呢？它们真的很恐怖，是不是？哦，还好吧。我们来看一个简单的例子，这应该能让你想起我们做过的一些Scala例子：

```
1:5 user=> (defn adder [constToAdd] #(+ constToAdd %1))
#'user/adder
1:6 user=> (def plus2 (adder 2))
#'user/plus2
1:7 user=> (plus2 3)
5
1:8 user=> 1:9 user=> (plus2 5)
7
```

上例中先定义了(adder)函数。这是一个构造其他函数的函数。如果你熟悉Java语言的工厂方法模式，可以把它当成Clojure的工厂方法实现。以其他函数为函数的返回值没什么好奇怪的，这是将函数作为普通值这一概念的重要体现。

这个例子给匿名函数用了缩写的#()形式。函数(adder)接受一个数值参数并返回一个函数，并且返回的是带一个参数的函数。

然后用(adder)定义了一个新形式：(plus2)。这个函数接受一个参数，并在这个参数上加2。这就是说绑定到(adder)内部的constToAdd的值是2。现在我们来构造一个新函数：

```
1:13 user=> (def plus3 (adder 3))
#'user/plus3
1:14 user=> (plus3 4)
7
1:15 user=> (plus2 4)
6
```

这段代码表明你还可以再构造其他函数(plus3)，绑定不同的值到constToAdd上。我们说函数(plus3)和(plus2)已经从它们所在的环境中捕获或“封装”了一个值[①]。需要注意的是(plus3)和(plus2)捕获的值是不同的，并且定义(plus3)对(plus2)捕获的值没有影响。

在自身环境内“封装”一些值的函数称为闭包，(plus2)和(plus3)就是闭包。在支持闭包的语言中，用一个制造者函数构造并返回另一个封装了一些东西的函数非常普遍。

接下来我们要讨论Clojure中一个强大的特性：序列。它们使用了跟Java的集合或迭代器类似的东西，但有些不同的属性。在代码中使用序列最能体现Clojure语言的力量，对于习惯了Java处理方式的程序员，Clojure的处理方式会让你耳目一新。

10.4　Clojure 序列

看下面这段代码中的Java迭代器。这是使用迭代器的老套路了。实际上，Java 5里的for循环在底层也会被转换成这种实现：

```
Collection<String> c = ...;

for (Iterator<String> it = c.iterator(); it.hasNext();) {
  String str = it.next();
  ...
}
```

对于简单集合的循环处理这就够了，比如Set或List。但Iterator接口只有next()和hasNext()方法，加上一个可选的remove()方法。

1. 残缺的Java迭代器

然而Java迭代器还有缺陷。迭代器接口所提供的集合交互方法满足不了需求。用Iterator只能做两件事：

- 查看集合中是否还有更多的元素；

① 此处的环境即指函数(adder)，而捕获的值即绑定到constToAdd的值。——译者注

- 取出下一个元素，并把迭代器向前推进。

Iterator最主要的问题是把取得下一个元素和向前推进合在了一起（如图10-5所示）。这意味着无法先对集合中的下一个元素进行检查，然后再决定它是需要特殊处理，还是完好无损地取出去。

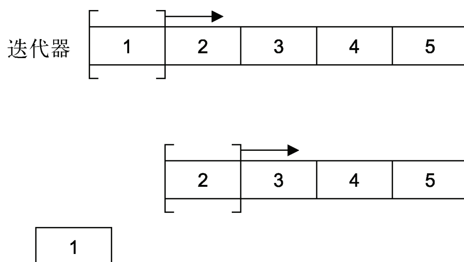


图10-5 Java迭代器的性质

从迭代器中取出下一元素的行为改变了它的状态。也就是说可变已经内建在Java处理集合和迭代器的方法中了，因此不可能用它构建出强健的多路解决方案。

2. Clojure的键抽象

Clojure采用了不同的方式。Clojure与Java中的集合与迭代器相对应的核心概念是序列（sequence），或者简称seq。它基本上是把两个Java类的一些特性集成到了一个概念里。这样做的动机有三个：

- 更强健的迭代器，特别是对于多路算法而言；
- 不可变能力，可以安全地在函数间传递序列；
- 实现懒序列的可能性（后面还会详细讨论）。

表10-4中列出了跟序列相关的一些核心功能。这些函数都不会改变它们的参数，如果它们需要返回不同的值，那会是一个不同的序列。

表10-4 基本的序列函数

函　　数	作　　用
`(seq <coll>)`	返回一个序列，作为所操作集合的“视图”
`(first <coll>)`	返回集合的第一个元素，如有必要，先在其上调用`(seq)`。如果集合为`nil`，则返回`nil`
`(rest <coll>)`	返回从集合中去掉第一个元素后得到的新序列。如果集合为`nil`，则返回`nil`
`(seq? <o>)`	如果o是一个序列则返回`true`（也就是实现了`ISeq`）
`(cons <elt> <coll>)`	在集合前面增加新元素，并返回由此得到的序列
`(conj <coll> <elt>)`	返回将新元素加到合适一端（向量的尾端和列表的头）的新集合
`(every? <pred-fn> <coll>)`	如果`(pred-fn)`对集合中的每个元素都返回逻辑真，则返回`true`

这里有几个例子：

```
1:1 user=> (rest '(1 2 3))
(2 3)
1:2 user=> (first '(1 2 3))
1
1:3 user=> (rest [1 2 3])
(2 3)
1:13 user=> (seq ())
nil
1:14 user=> (seq [])
nil
1:15 user=> (cons 1 [2 3])
(1 2 3)
1:16 user=> (every? is-prime [2 3 5 7 11])
true
```

有一点要重点关注一下，列表是自身的序列，而向量不是。因此从理论上来说，不能在向量上调用(rest)。可实际上是可以的，因为(rest)在操作向量之前先在其上调用了(seq)。这是序列结构中普遍存在的属性：很多序列函数都会接受比序列更通用的对象，并在开始之前先调用(seq)。

我们在这一节中准备探索seq的一些基本属性和用法，尤其会重点关注懒序列和变参函数。其中第一个概念“懒”，是Java中不太会涉及的编程技术[①]，所以对你来说它可能比较新颖。现在我们就来看一下吧。

10.4.1　懒序列

在编程语言里，懒是一个强大的概念。其基本思想是将表达式的计算推迟到需要时。体现在Clojure中就是序列可以不是完整的值列表，其中的值可以在被请求时取得（比如根据需要通过调用函数生成它们）。

在Java中，要满足这样的想法就得靠定制的List实现，而且要写大量的套路化代码才可能实现。用Clojure中的宏只要做一点儿工作就能创建出懒序列。

想一想怎么才能创建出一个懒惰的、可能包含无限数量值的序列。很明显，用函数来生成序列内的元素。这个函数应该做两件事：

- 返回序列中的下一个元素；
- 接受数量固定、有限的参数。

数学家会说这样一个函数定义的是递归关系，并且这样的关系用递归的方式处理再恰当不过了。

假设有一台在栈空间和其他能力上都不受限制的机器，并且可以执行两个线程：一个用来生成无限的序列，另外一个使用该序列。那我们就可以在生成线程里用递归定义懒序列，类似下面这段伪代码：

```
(defn infinite-seq <vec-args>
(let [new-val (seq-fn <vec-args>)]
  (cons new-val (infinite-seq <new-vec-args>))))
```

① 用过Hibernate的人一定知道懒加载（因为它原来经常爆异常），其基本思路“延迟”跟懒是一样的。——译者注

实际上在Clojure中这是行不通的，因为(infinite-seq)上的递归会导致栈溢出。但要是加上一个结构，告诉Clojure不要疯狂递归，仅根据需要进行处理，是可以做到的。

不仅如此，你还能在一个线程内做到这一点，如下例所示。代码清单10-5中为某个数k定义了懒序列k，k+1，k+2，...。

代码清单10-5　懒序列的例子

```
(defn next-big-n [n] (let [new-val (+ 1 n)]
  (lazy-seq                                          ; lazy-seq标记
    (cons new-val (next-big-n new-val))              ; 无限递归
 )))

(defn natural-k [k]
  (concat [k] (next-big-n k)))                       ; concat限制递归

1:57 user=> (take 10 (natural-k 3))
(3 4 5 6 7 8 9 10 11 12)
```

(lazy-seq)形式是关键，它标记了发生无限递归的点，还有(concat)，可以安全地处理递归。然后你就可以用(take)形式从懒序列中取出所需的元素了，这个基本上是用(next-big-n)形式定义的。

懒序列是极其强大的特性，实践会告诉你它们是Clojure军火库中的强大武器。

10.4.2　序列和变参函数

Clojure函数有一个强大的特性，它天生就具备参数数量可变的能力，有时称为函数的*变元*（arity）。参数数量可变的函数称为*变参函数*（variadic）。

代码清单10-1中讨论过的函数(const-fun1)可以作为一个简单的例子。这个函数接受一个参数并抛弃它，总是返回值1。请看传入多个参数给(const-fun1)时会发生什么：

```
1:32 user=> (const-fun1 2 3)
java.lang.IllegalArgumentException: Wrong number of args (2) passed to:
user$const-fun1 (repl-1:32)
```

Clojure编译器仍然会对传给(const-fun1)的参数数量（和类型）做一些检查。对于简单地抛弃所有参数并返回一个常量值的函数来说，这似乎过于严格了。在Clojure中能接受任意数量参数的函数看起来会是什么样的呢？

代码清单10-6展示了如何实现一个这样的(const-fun1)常量函数。我们管它叫(const-fun-arity1)，变元的const-fun1。这是在Clojure标准函数库中(constantly)函数的自产版。

代码清单10-6　带有变元的函数

```
1:28 user=> (defn const-fun-arity1
  ([] 1)
  ([x] 1)                                 ; 带不同签名的多个defn
  ([x & more] 1)
 )
#'user/const-fun-arity1
```

```
1:33 user=> (const-fun-arity1)
1
1:34 user=>  (const-fun-arity1 2)
1
1:35 user=>  (const-fun-arity1 2 3 4)
1
```

这个函数的定义不是一个参数向量后跟着函数行为的定义。而是有一系列这种组合，每个组合里都是一个参数向量（构成了这一版本函数的有效签名）和这一版本函数的实现。

这跟Java的方法重载类似。传统做法一般是定义几个特殊情况下的形式（没有参数、一个或两个参数）和最后一个参数为序列的额外形式。代码清单10-6中就是参数向量为`[x & more]`的那个。`&`符号表明这是该函数的变参版本。

序列是Clojure的创新。实际上，用Clojure编程主要就是要思考怎么用序列解决特定问题。

Clojure的另一项重要创新是Clojure和Java的集成，也就是我们下一节的主题。

10.5　Clojure 与 Java 的互操作

Clojure从一开始就设计为JVM语言，并且不会对程序员完全隐藏JVM特性。这些特殊的设计在几个地方都有体现。比如在类型系统层面，Clojure的列表和向量都实现了Java集合类库中的标准接口`List`。另外，Clojure使用Java的类库非常容易，反之亦然。

这意味着Clojure程序员可以使用Java中丰富的类库和工具，以及JVM的性能和其他特性。这一节会涉及这种互操作性的几方面内容，特别是：

- 从Clojure中调用Java；
- Java如何见到Clojure函数的类型；
- Clojure代理；
- 用REPL做探索性编程；
- 从Java中调用Clojure。

我们先看看从Clojure中如何访问Java方法，开始它们的集成探索之旅吧。

10.5.1　从Clojure中调用Java

看一下这段在REPL中进行计算的Clojure代码：

```
1:16 user=> (defn lenStr [y] (.length (.toString y)))
#'user/lenStr
1:17 user=> (schwartz ["bab" "aa" "dgfwg" "droopy"] lenStr)
("aa" "bab" "dgfwg" "droopy")
1:18 user=>
```

这段代码用Schwartzian转换对一个字符串向量排序，排序标准是字符串的长度。其中用到了形式`(.toString)`和`(.length)`，这都是Java方法，它们是在Clojure对象上调用的。符号开始部分的句号`.`表示运行时应该在下一个参数上调用该名称的方法，底层是用`(.)`宏实现的。

所有用`(def)`或它的变体定义的Clojure值都被放在`clojure.lang.Var`实例中，它可以承载

任何java.lang.Object，所以任何可以在java.lang.Object调用的方法都可以在Clojure值上调用。另外一些跟Java交互的形式是用来调用静态方法的

```
(System/getProperty "java.vm.version")
```

（此处是调用System.getProperty()）和用于访问静态公共变量（比如常量）的

```
Boolean/TRUE
```

在后面两个例子中已经用到了Clojure命名空间的概念。跟Java包的概念类似，并且常用的Java包都有对应的映射缩写形式，比如前面那些。

Clojure调用的本质

Clojure中的函数调用实际上是JVM的方法调用。JVM不能保证像类Lisp语言（特别是Scheme）通常做的那样优化掉尾递归。JVM上一些其他的Lisp方言觉得它们需要真正的尾递归，因此不准备把Lisp函数调用跟JVM方法调用完全等同起来。而Clojure完全以JVM为平台，甚至不惜违背通常的Lisp实践。

如果你想创建一个新的Java对象实例并在Clojure中操作它，用(new)形式就可以轻松做到。它还有个备选的缩写形式，在类名之后跟一个句号，可以归结为(.)宏的另一个用法：

```
(import '(java.util.concurrent CountDownLatch LinkedBlockingQueue))
(def cdl (new CountDownLatch 2))
(def lbq (LinkedBlockingQueue.))
```

这里还用了(import)形式，只用一行就可以导入一个包的很多Java类。

我们在前面提过，Clojure的类型系统有些地方跟Java是一致的，我们来看看其中的细节。

10.5.2　Clojure值的Java类型

从REPL中很容易看到某些Clojure值的Java类型：

```
1:8 user=> (.getClass "foo")
java.lang.String

1:9 user=> (.getClass 2.3)
java.lang.Double
1:10 user=> (.getClass [1 2 3])
clojure.lang.PersistentVector
1:11 user=> (.getClass '(1 2 3))
clojure.lang.PersistentList
1:12 user=> (.getClass (fn [] "Hello world!"))
user$eval110$fn__111
```

首先要看到所有Clojure值都是对象，JVM的原始类型默认情况下是不对外的（尽管从性能角度来看有办法得到原始类型）。如你所料，字符串和数字值直接映射到对应的Java引用类型上去了（java.lang.String、java.lang.Double等）。

匿名的"Hello world!"函数的名字表明它是一个动态生成类的实例。这个类会实现

clojure.lang.IFn接口，Clojure用该接口表明这个值是个函数，你可以把它当做java.util.concurrent里的Callable接口。

序列会实现clojure.lang.ISeq接口。它们通常是抽象类ASeq或懒实现LazySeq的具体子类。

我们已经看过几种值的类型了，但这些值是怎么保存的呢？就像我们在本章一开始提到的，(def)把符号绑到一个值上，这样会创建一个var。这些var是clojure.lang.Var类型（它所实现的接口中也有IFn）的对象。

10.5.3　使用Clojure代理

Clojure有一个强大的宏(proxy)，你可以用它创建扩展Java类（或实现接口）的Clojure对象。比如代码清单10-7重新实现了之前的一个例子（代码清单4-13），由于Clojure语法更加紧凑，所以这个例子的核心代码只有一点。

代码清单10-7　重温调度执行者

```
(import '(java.util.concurrent Executors LinkedBlockingQueue TimeUnit))
(def stpe (Executors/newScheduledThreadPool 2))       ; STPE工厂方法
(def lbq (LinkedBlockingQueue.))

(def msgRdr (proxy [Runnable] []                      ; 定义匿名的Runnable实现
  (run [] (.toString (.poll lbq)))
 ))

(def rdrHndl
➥  (.scheduleAtFixedRate stpe msgRdr 10 10 TimeUnit/MILLISECONDS))
```

(proxy)的一般形式是：

```
(proxy [<超类/接口>] [<args>] <命名函数的实现>+)
```

第一个向量参数是这个代理类应该实现的接口。如果这个代理还要扩展Java类（如果可以的话，当然，只能扩展一个Java类），这个类名必须是向量中的第一个元素。

第二个向量参数包含传给超类构造方法的参数。这个向量经常是空的，并且如果(proxy)形式只是实现Java接口的话，那它肯定是空的。

这两个参数之后是一个或多个表示单个方法实现的形式，按接口的要求或超类指定的实现。

用(proxy)形式可以做出任何Java接口的简单实现。这促成了一种吸引人的可能性：用Clojure REPL作为实验Java和JVM代码的扩展游戏床。

10.5.4　用REPL做探索式编程

探索式编程的核心思想是减少要编写的代码量，因为Clojure的语法和REPL提供的实时互动环境，REPL不仅是探索Clojure编程的理想环境，也是学习Java类库的极佳选择。

我们来看一下Java列表实现。它们都有返回Iterator类型对象的iterator()方法。但

Iterator是个接口，所以你可能对真正的实现类型感到好奇。用REPL很容易找出答案：

```
1:41 user=> (import '(java.util ArrayList LinkedList))
java.util.LinkedList
1:42 user=> (.getClass (.iterator (ArrayList.)))
java.util.ArrayList$Itr
1:43 user=> (.getClass (.iterator (LinkedList.)))
java.util.LinkedList$ListItr
```

(import)形式从java.util包中导入了两个类。然后在REPL内用Java的getClass()方法。可以看到迭代器实际上是内部类提供的。也许你不应该对此感到吃惊，因为我们在10.4节讨论过，迭代器和它们的集合绑定很紧密，所以它们也许需要了解这些集合的内部实现细节。

在前面这个例子中值得注意的是，我们一个Clojure结构也没用，只用了一点语法。我们操作的所有东西实际上都是Java结构。尽管如此，我们还是假设你想用不同的方式，在Java程序里用Clojure。下一节将会向你展示如何实现这一目的。

10.5.5 在Java中使用Clojure

Clojure的类型系统跟Java高度一致。Clojure数据结构全是真正的Java集合，都实现了对应接口的所有必需部分。因为接口的可选部分一般都跟修改数据结构有关，而Clojure数据结构不可变，所以一般都没实现。

类型系统的一致性使得在Java程序里使用Clojure数据结构成为可能。Clojure自身的性质加强了这种可行性——它是采用调用机制的JVM编译型语言。这最大限度地减少了运行时的问题，意味着从Clojure中得到的类几乎跟其他任何Java类一样。解释型语言跟Java的互操作会更加困难，并且通常需要最基本的非Java语言运行时支持。

下面这个例子展示了Clojure的seq结构如何用在一个普通的Java字符串中。要运行这段代码，需要把clojure.jar放在classpath上：

```
ISeq seq = StringSeq.create("foobar");
while (seq != null) {
  Object first = seq.first();
  System.out.println("Seq: "+ seq +" ; first: "+ first);
  seq = seq.next();
}
```

上面的代码使用了StringSeq类中的工厂方法create()。它给出了字符串中字符序列的seq视图。first()和next()方法返回新值，而不是修改已有的seq，就跟我们在10.4节讨论的一样。

截止目前我们只是在处理单线程的Clojure代码。下一节我们要谈论Clojure中的并发。特别是Clojure对状态和可变性的处理方式，这使得它的并发模型跟Java的差别很大。

10.6 Clojure 并发

Java的状态模型从根本上来说是基于对象可变思想的。正如第4章中所提及的，这会直接导致并发代码的安全问题。在某一线程修改对象的状态时，为了防止其他线程看到对象的中间（即不一致）状态，需要引入相当复杂的锁策略。这些策略理解难，调试难，测试更难。

Clojure的并发概念在某些方面不像Java中那么底层。比如说，由Clojure运行时管理线程池的使用（开发人员在这方面几乎或根本不能控制）看起来可能有点奇怪。但是让平台（此处即Clojure运行时）细致地做好内务工作的好处在于，开发人员可以专注于更重要的任务，比如总体设计。

Clojure的指导思想是默认把线程彼此隔开，这种实现并发安全的办法由来已久。假定“没有共享资源”的基线和采用不可变值使Clojure避开了很多Java所面临的问题，从而可以专注于为并发编程安全地共享状态的方法。

注意　为了帮助提升安全性，Clojure的运行时提供了线程协调机制，我们强烈建议你使用这些机制，而不是用Java的惯例或构造自己的并发结构。

实际上，Clojure用不同的方法实现了不同的并发模型：未来式（future）、并行调用（pcall）、引用形式（ref）和代理（agent）。且听我们一一道来，先从最简单的开始。

10.6.1　未来式与并行调用

第一个也是最明显的一个状态分享办法就是不分享。实际上，我们一直使用的Clojure结构var本质上是不可以共享的。如果两个不同的线程继承了名字相同的var，并在线程里重新绑定了它，那绑定只在这些线程内部可见，绝不可能被其他线程共享。

可以利用Clojure跟Java的紧密结合启动新线程，也就是说在Clojure中写Java并发代码非常容易。但其中有些抽象在Clojure中有更干净的形式。比如对于第4章介绍的Java未来式（`Future`），Clojure提供了非常干净的方式。代码清单10-8是个简单的例子。

代码清单10-8　Clojure中的`Future`

```
user=> (def simple-future
 (future (do
   (println "Line 0")
   (Thread/sleep 10000)
   (println "Line 1")
   (Thread/sleep 10000)
   (println "Line 2"))))
#'user/simple-future
Line 0                                          ◁── 马上开始执行
user=> (future-done? simple-future)
user=> false
Line 1
user=> @simple-future                           ◁── 解引用导致阻塞
Line 2
nil
user=>
```

这段代码用`(future)`建立了一个`Future`。创建之后它马上就开始在后台线程中运行，所以在Clojure REPL中看到了输出`Line 0`（然后是`Line 1`）——代码已经开始在另一个线程上运行了。接着可以用`(future-done?)`来检查代码是否已经运行完，这个调用是非阻塞的。然而对

future的解引用会阻塞调用线程，直到函数完成。

这实际上是Clojure对Java Future的一个瘦封装，语法更干净。Clojure还提供了对并发程序员非常有帮助的辅助形式。有个简单的函数是(pcalls)，可以接受数量可变的零参函数，让它们并发执行。它们在运行时管理的线程池上执行，并返回一个懒序列结果。试图访问序列中的任何还没完成的元素会导致访问线程被阻塞。

代码清单10-9建立了一个单参函数(wait-with-for)。它用了一个类似10.3.2节介绍过的loop形式。可以用它创建一些零参函数(wait-1)、(wait-2)等，并把它们传给(pcalls)。

代码清单10-9 Clojure中的并行调用

```
user=> (defn wait-with-for [limit]
  (let [counter 1]
    (loop [ctr counter]
      (Thread/sleep 500)
      (println (str "Ctr=" ctr))
      (if (< ctr limit)
          (recur (inc ctr))
          ctr))))
#'user/wait-with-for
user=> (defn wait-1 [] (wait-with-for 1))
user=> #'user/wait-1
user=> (defn wait-2 [] (wait-with-for 2))
user=> #'user/wait-2
user=> (defn wait-3 [] (wait-with-for 3))
user=> #'user/wait-3
user=> (def wait-seq (pcalls wait-1 wait-2 wait-3))
#'user/wait-seq
Ctr=1
Ctr=1
Ctr=1
Ctr=2
Ctr=2
Ctr=3

user=> (first wait-seq)
1
user=> (first (next wait-seq))
2
```

因为线程睡眠值只有500毫秒，等待函数很快就能完成。通过调整超时（比如延迟到10秒），很容易验证由(pcalls)返回的懒序列wait-seq是否有上面描述的那种阻塞行为。

对于不需要共享状态的情况，这种简单的多线程结构挺好，但在很多应用中，不同的处理线程都要在运行过程中相互通信。Clojure有几个模型可以处理这种情况，接下来我们先看看其中的一个：借助(ref)形式实现的状态共享。

10.6.2 ref形式

ref是Clojure在线程间共享状态的办法。它们基于运行时提供的一个模型，在这个模型中，状

态的改变要能被多个线程见到。该模型在符号和值之间引入了一个额外的中间层。也就是说，符号绑定到值的引用上，而不是直接绑到值上。这个系统基本上是事务化的，并且由Clojure运行时进行协调。如图10-6所示。

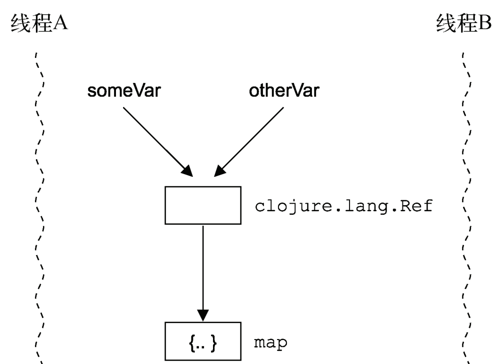


图10-6　软件事务内存

这一中间层意味着改变或更新ref之前必须把它放在一个事务中。当事务完成的时候，或者全变了，或者什么也没变。这跟数据库中的事务是类似的。

这可能有点抽象了，所以我们来看一个模拟ATM的例子。在Java中，要对所有敏感数据加锁保护。代码清单10-10是一个简单的自动提款机模型，包括锁。

代码清单10-10　Java中的ATM模型

```
public class Account {
  private double balance = 0;
  private final String name;
  private final Lock lock = new ReentrantLock();

  public Account(String name_, double initialBal_){
    name = name_;
    balance = initialBal_;
  }

  public synchronized double getBalance(){
    return balance;
  }

  public synchronized void debit(double debitAmt_) {
    balance -= debitAmt_;
  }

  public String getName() {
    return name;
  }

  public String toString() {
    return "Account [balance=" + balance + ", name=" + name + "]";
  }
```

```
  public Lock getLock() {
    return lock;
  }
}

public class Debitter implements Runnable {
  private final Account acc;
  private final CountDownLatch cdl;

  public Debitter(Account account_, CountDownLatch cdl_) {
    acc = account_;
    cdl = cdl_;
  }

  public void run() {
    double bal = acc.getBalance();
    Lock lk = acc.getLock();

    while (bal > 0) {
      try {
        Thread.sleep(1);
      } catch (InterruptedException e) { }
      lk.lock();                              // 能在acc上同步
      bal = acc.getBalance();                 // 必须重新取得余额
      if (bal > 0) {
        acc.debit(1);
        bal--;
      }
      lk.unlock();
    }
    cdl.countDown();
  }
}

Account myAcc = new Account("Test Account", 500 * NUM_THREADS);
CountDownLatch stopl = new CountDownLatch(NUM_THREADS);

for (int i=0; i<NUM_THREADS; i++) {
  new Thread(new Debitter(myAcc, stopl)).start();
}
stopl.await();
System.out.println(myAcc);
```

再来看看用Clojure怎么写。先来个单线程版本。然后我们再开发一个并发版本跟单线程版本比较，这样并发代码应该更容易理解。

代码清单10-11是单线程版本。

代码清单10-11　Clojure中的简单ATM模型

```
(defn make-new-acc [account-name opening-balance]
  {:name account-name :bal opening-balance})

(defn loop-and-debit [account]
  (loop [acc account]
    (let [balance (:bal acc) my-name (:name acc)]
```

```
      (Thread/sleep 1)
      (if (> balance 0)
        (recur (make-new-acc my-name (dec balance)))
        acc
      ))))

(loop-and-debit (make-new-acc "Ben" 5000))
```

用循环/递归代替**Java**中的**while**

这段代码跟Java版比起来非常紧凑。必须承认，这是单线程的，但还是比Java的代码少了很多。运行代码会得到期望的结果：一个余额为0的`acc`映射。现在我们看看并发形式。

要让这段代码并行，需要引入ref。它们是用`(ref)`形式创建的，并且类型为`clojure.lang.Ref`的JVM对象。通常建立时会带一个保存状态的映射，此外还需要`(dosync)`形式来设置事务。在事务之内，还要用到`(alter)`形式来修改ref。使用ref的多线程ATM函数如代码清单10-12所示。

代码清单10-12　多线程ATM

```
(defn make-new-acc [account-name opening-balance]
  (ref {:name account-name :bal opening-balance}))

(defn alter-acc [acc new-name new-balance]
  (assoc acc :bal new-balance :name new-name))

(defn loop-and-debit [account]
  (loop [acc account]
    (let [balance (:bal @acc)
          my-name (:name @acc)]
      (Thread/sleep 1)
      (if (> balance 0)
        (recur (dosync (alter acc alter-acc my-name (dec balance)) acc))
        acc
      ))))

(def my-acc (make-new-acc "Ben" 5000))

(defn my-loop [] (let [the-acc my-acc]
  (loop-and-debit the-acc)
 ))

(pcalls my-loop my-loop my-loop my-loop my-loop)
```

必须返回值，而不是引用

就像注释中说的，对值进行操作的`(alter-acc)`函数必须返回一个值。所操作的值是对当前事务中线程可见的本地值，这称为*事务内的值*。返回的值是在变更函数返回之后的ref新值。在退出`(dosync)`所定义的事务块之前，这个值对外界是不可见的。

与此同时，其他事务可能像这个一样也在进行。如果是这样，Clojure STM系统会进行跟踪，并且只允许那些自开始以来已经提交过的事务组成的事务提交。如果不一致，它会回滚，并且可能在得到更新过的状态后再次尝试。

如果事务做了任何会产生副作用的事情（比如日志文件或其他输出），这个重试行为可能会引发问题。让事务化部分在函数式编程中（即没有副作用）尽可能地保持简单纯粹是你的责任。

对于某些多线程方式而言，这种持乐观态度的事务行为看起来可能是相当重量级的做法。有些并发应用只需偶尔在线程间进行通信，并且是以相当不对称的风格。幸运的是，Clojure提供了另外一种更好地体现“过后就忘“原则的并发机制，这也是我们下一节的主题。

10.6.3 代理

代理是Clojure中异步的、面向消息的并发原语。Clojure代理不是共享状态，而是属于另外一个线程的一点儿状态，但它会从另外一个线程中接收消息（以函数的形式）。这乍看起来可能是个奇怪的想法，尽管遇到过Scala的actor之后这种感觉可能会少一点。

> “我离它们太远了，只能把礼物装进包裹寄给它们，”她想，“这也太滑稽了，给自己的双脚送礼物还需要邮寄！地址写起来就更有趣了！”
>
> ——《爱丽丝梦游仙境》，刘易斯·卡罗尔

应用到代理上的函数在代理的线程上运行。这个线程是由Clojure运行时管理的，在一个程序员通常无法访问的线程池里。运行时还会保证代理中那些可以被外界看到的值是孤立的和原子的。这就是说用户代码只会见到状态修改之前或之后的代理值。

代码清单10-13是个简单的代理例子，跟用来讨论future的例子类似。

代码清单10-13 Clojure代理

```
(defn wait-and-log [coll str-to-add]
  (do (Thread/sleep 10000)
    (let [my-coll (conj coll str-to-add)]
      (Thread/sleep 10000)
      (conj my-coll str-to-add))))

(def str-coll (agent []))

(send str-coll wait-and-log "foo")

@str-coll
```


send调用派发了一个(wait-and-log)调用给代理，通过使用REPL解引用，结果就像承诺的那样，你绝不会看到代理的中间状态——只有最后的状态出现了（字符串"foo"被添加了两次）。

实际上，代码清单10-13上的(send)调用很容易让人联想到爱丽丝的脚的地址。刘易斯·卡罗尔很可能是用Clojure代码写的地址：

```
爱丽丝的右脚收
    壁炉前的毛毯上
      靠近挡板
       (带去爱丽丝的爱)
```

在你认为一个人的脚是身体的有机组成时，这的确挺怪异的。同样，发消息给Clojure管理的线程池中一个线程上的代理看起来也挺怪异的，两个线程还共享一个地址空间。但你目前多次遇到的一个并发主题就是如果它能让用法更加简单清晰，额外的复杂性可能是件好事。

10.7　小结

作为一门语言，Clojure可以说是我们见过的几门语言中跟Java差别最大的。它对Lisp的传承、对不可变性的强调以及独特的编程方式，让它看起来变成了完全独立的语言。但它和JVM的紧密结合、与类型系统的一致性（即便它提供了序列等替代方案），还有探索式编程的能力，让它成为与Java互补性非常强的一门语言。

任何地方的协同都没有Clojure运行时对线程和并发底层特性的代理控制更清晰。这让程序员可以放手去关注多线程的设计和高层问题。这就跟Java的垃圾收集设施可以让你无需关心内存管理的细节一样。

本部分研究的不同语言间的差别展示了Java平台的进化能力，并且证明了它仍然是应用开发的理想目标。这也是对JVM灵活性和性能的证明。

在本书的最后一部分，我们会向你展示三门新语言为软件工程实践提供的新方式。下一章全部是关于测试驱动开发的内容——你在Java世界中很可能已经碰到过这一主题了。但Groovy、Scala和Clojure提供了全新的视角，有望巩固和加强你已经知道的那些东西。

Part 4 第四部分

多语种项目开发

在最后一部分，我们会把已经学到的平台和多语言编程知识应用到现代软件开发中最常见和最重要的技术上。

要成为一名优秀的 Java 开发人员，不仅仅是掌握 JVM 和它上面跑的语言那么简单。要成功交付软件，还要遵循业界最佳实践。幸好，这些实践中有相当一部分是从 Java 生态系统中开始的，所以我们有很多东西可以聊。

我们会用一整章的内容讨论测试驱动开发（TDD）的基础知识，以及如何把测试概念应用到极其复杂的测试场景中。另一章会集中讨论如何将正规的构建生命周期引入构建流程中，包括持续集成技术。这两章会介绍一些工具，比如用于测试的 JUnit、用于构建的 Maven，以及用于持续集成的 Jenkins。

我们还会讨论 Java 7 时代的 Web 开发，会涉及为项目选择最适合框架的标准，还有如何在这个环境中快速开发。

如果你看过第三部分，应该了解非 Java 语言在 TDD、构建生命周期和快速 Web 开发领域都有举足轻重的作用。无论是用于 TDD 的 ScalaTest 框架，或者用于构建 Web 应用的 Grails（Groovy）和 Compojure（Clojure）框架，Java/JVM 生态系统中的很多方面都受到了这些新语言的影响。

我们会向你展示如何把新语言的力量作用到你所熟悉的软件开发工艺上。与 JVM 坚实的基础和 Java 生态系统结合为一个整体，你会发现那些接受多语言观点的开发人员可能会收获颇丰。

最后一章我们会看一看平台的未来，并预测一下将来。第四部分全是前沿内容，所以现在就让我们翻开新的一页，向着地平线推进吧！

第 11 章

测试驱动开发

本章内容

- ❑ 实行测试驱动开发的好处
- ❑ TDD的核心：红—绿—重构循环周期
- ❑ JUnit，公认的Java测试框架
- ❑ 四种测试替身：虚设、伪装、存根和模拟
- ❑ 用内存数据库测试DAO代码
- ❑ 用Mockito模拟子系统
- ❑ 使用Scala测试框架ScalaTest

测试驱动开发（TDD）进入软件开发行业已经有相当长的时间了。它的基本前提是在编写真正的功能实现代码之前先写测试代码，然后根据需要重构实现代码。比如要写一段拼接两个`String`对象（`"foo"`和`"bar"`）的实现代码，应该先写测试代码（测试结果必须等于`"foobar"`），以确保你能判断实现是否正确。

很多开发人员都知道JUnit，也会在开发时不定期用到它。但他们一般是写完实现代码之后才编写测试代码，因此体会不到TDD的益处。

尽管TDD的概念看起来非常普及，但实际上很多开发人员并不清楚为什么要采用TDD。对于很多开发人员来说，“为什么要写测试驱动代码以及有什么好处”一直是个问题。

我们认为消除恐惧和不确定性是编写测试驱动代码的重要原因。Kent Beck（JUnit测试框架的发明人之一）在*Test-Driven Development: by Example*①（Addison-Wesley Professional，2002）一书中对此总结得很好：

- ❑ 恐惧会让你小心试探；
- ❑ 恐惧会让你尽量减少沟通；
- ❑ 恐惧会让你羞于得到反馈；
- ❑ 恐惧会让你脾气暴躁。

TDD可以祛除恐惧，让优秀的Java开发者变得更加自信、善于沟通、乐于接受并更加快乐。

① 中文版《测试驱动开发》已由中国电力出版社于2004年出版。——编者注

换句话说，TDD能帮你摆脱下面这种心态：

- ❑ 在开始新工作时，“我不知道从哪里开始，所以只好将就着做一些修改”；
- ❑ 在修改已有代码时，“我不知道现有代码怎么运行，所以我私下认为不能碰它们”。

TDD带来的很多好处并不会马上显现：

- ❑ 更清晰的代码——只写需要的代码；
- ❑ 更好的设计——有些开发人员管TDD叫测试驱动的设计；
- ❑ 更出色的灵活性——TDD鼓励按接口编码；
- ❑ 更快速的反馈——不会直到系统上线才知道bug的存在。

刚入门的开发人员有时认为TDD不是“普通”开发人员用的技术，这是他们采用TDD的一个障碍。他们的感觉是只有那些想象中的“敏捷派”或其他神秘组织的成员才会用TDD。这种认识完全错误，我们会在后面解释。TDD是给所有开发人员使用的技术。

另外，敏捷和软件工艺运动都是为了让开发人员活得更轻松。它们肯定不会拒绝别人使用TDD或其他任何技术。

本章首先解释TDD背后的基本思想红—绿—重构循环，然后介绍Java测试框架中的主力JUnit，并用一个简单的例子来阐明其原则。

敏捷宣言和软件工艺运动

敏捷运动（http://agilemanifesto.org/）已经开展很长时间了，可以说部分改善了软件开发行业。很多伟大的技术，比如TDD，都是这项运动所倡导的。软件工艺是一项新运动，鼓励参与者编写清晰的代码（http://manifesto.softwarecraftsmanship.org/）。

我们喜欢取笑实行敏捷和软件工艺运动的弟兄们。可是，我们自己甚至也拥护它（大多数时候都是如此）。但优秀的Java开发人员，请不要忽视那些对你有用的东西。TDD是一项软件开发技术，仅此而已。

接下来，我们会介绍TDD使用的四大类伪装对象。它们能简化受试代码和第三方类库中代码的隔离，或隔离数据库之类的子系统行为，所以它们很重要。随着依赖项变得越来越复杂，伪装对象也要变得越来越聪明。最终我们会介绍模拟和Mockito类库，它是一个流行的模拟工具，可以让开发人员在不受外部系统影响的环境下进行测试。

开发人员非常熟悉Java测试框架（特别是JUnit），并且一般都有用它们编写测试代码的经验。但对于如何用测试驱动Scala、Clojure等新语言，你可能毫无头绪。因此我们会介绍Scala测试框架ScalaTest，以确保你能在开发Scala代码时应用TDD。

让我们开始了解这个有点奇怪的TDD吧。

11.1 TDD 概览

TDD可以应用在多个层级上。表11-1列出了通常会采用TDD的四个测试层级。

表11-1　TDD的测试层级

层　　级	描　　述	例　　子
单元测试	通过测试验证一个类中包含的代码	测试BigDecimal类中的方法
集成测试	通过测试验证类之间的交互	测试Currency类以及它如何跟BigDecimal交互
系统测试	通过测试验证运行的系统	从UI到Currency类测试会计系统
系统集成测试	通过测试验证运行的系统，包括第三方组件	测试会计系统，包括它与第三方报表系统间的交互

在单元测试中使用TDD是最容易的，如果你对TDD不熟悉，这一层就是个很好的起点。本节主要讲述如何在单元测试层中使用TDD。后续章节会讨论其他层级，包括第三方组件和子系统的测试。

提示　处理没有或只有很少测试的遗留代码是个恐怖的任务。我们几乎不可能把所有测试都追加上，因此，应该只是为添加的新功能加上测试代码。请参阅Michael Feathers的*Working Effectively with Legacy Code*①（Prentice Hall，2004）获取更多帮助。

我们一开始会简单介绍一下TDD的基本前提——红—绿—重构循环——用JUnit测试计算剧院门票销售收入的代码②。只要遵照红—绿—重构循环，基本上就可以使用TDD！之后我们会探究一下红—绿—重构循环背后的思想，让你对为什么应该采用这种技术有更清楚地认识。最后我们将介绍JUnit这个公认的Java开发者测试框架，讲解它的基本用法。

让我们开始吧，先来一个TDD三步（红—绿—重构）测试计算剧院门票销售收入的实际例子。

11.1.1　一个测试用例

如果你有TDD方面的经验，可以自行决定是否跳过这一节，不过这个小例子中有些新东西。假定有人要你写一个坚若磐石的方法来计算剧院门票的销售收入。剧院会计最初给出的业务规则很简单：

- 门票的底价是30美元；
- 总收入=售出票数*价格；
- 剧院有100个座位。

因为剧院工作人员不懂软件，所以他们现在还必须手工录入门票的销售数量。

如果你做过TDD，应该知道它的三个基本步骤：红、绿、重构。如果刚接触TDD，或者想复习一下，那就请看一下Kent Beck在《测试驱动开发》中对这些步骤的定义：

---

① 中文版《修改代码的艺术》已由人民邮电出版社于2007年出版（更多信息请参见http://www.ituring.com.cn/book/536）。——编者注

② 销售剧院门票在我的家乡伦敦是个大生意，最起码在我们写这本书的时候是。

(1) 红，写一些不能用的测试代码（失败测试）;

(2) 绿，尽快让测试通过（通过测试）;

(3) 重构，消除重复（经过细化的通过测试）。

为了让你了解TicketRevenue应该达到什么效果，请先看一下这些伪代码。

```
estimateRevenue(int numberOfTicketsSold)

if (numberOfTicketsSold is less than 0 OR greater than 100)
then
  Deal with error and exit
else
  revenue = 30 * numberOfTicketsSold;
  return revenue;
endif
```

注意，千万别太深入。测试最终会驱动设计，也会部分影响实现。

> **注意**　我们在11.1.2节会涉及开始失败测试的办法，但在这个例子中我们准备写一个甚至还无法编译的测试！

接下来我们先用JUnit写一个失败单元测试。如果你不了解JUnit，请跳到11.1.4节，然后再回来。

1. 编写失败测试（红）

这一步的要点是以一个会失败的测试开始。实际上，这个测试甚至无法编译，因为你还没有TicketRevenue类！

在跟会计开过一个简短的白板会议后，你意识到测试代码需要覆盖五种情况：售票数量为负数、0、1、2~100，还有大于100。

> **提示**　编写测试代码（特别是牵扯到数值时）有一个很好的经验法则，要考虑值为0/null、1和很多（*N*）的情况。再进一步考虑*N*上的其他限制，比如数量为负或超出上限。

我们决定先写一个测试覆盖销售一张门票收入的情况。测试代码看起来应该如代码清单11-1所示（记住这个阶段不用编写完美的通过测试）。

代码清单11-1　为TicketRevenue编写的失败单元测试

```
import java.math.BigDecimal;
import static junit.framework.Assert.*;
import org.junit.Before;
import org.junit.Test;
public class TicketRevenueTest {

  private TicketRevenue venueRevenue;
  private BigDecimal expectedRevenue;

  @Before
  public void setUp() {
    venueRevenue = new TicketRevenue();
  }
```

```
    @Test
    public void oneTicketSoldIsThirtyInRevenue() {        <-- 销售一张票的情况
      expectedRevenue = new BigDecimal("30");
      assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(1));
    }
}
```

测试期望销售一张门票得到的收入等于30。

但这个测试不能编译，因为有`estimateTotalRevenue(int numberOfTicketsSold)`方法的`TicketRevenue`类还不存在呢。为了运行测试，可以先随便写一个让测试可以编译的实现。

```
public class TicketRevenue {
  public BigDecimal estimateTotalRevenue(int i) {
      return BigDecimal.ZERO;
  }
}
```

现在测试代码能编译了，你可以在自己喜欢的IDE中运行它。每种IDE都有自己运行JUnit测试的办法，但一般都能在选中测试类后，从右键弹出菜单中选择运行测试。一旦运行，IDE一般都会更新窗口告诉你测试失败了，因为你所期望的30和`estimateTotalRevenue(1);`返回的值不符，它的返回值是0。

失败测试有了，接下来该做通过测试了（变绿）。

2. 编写通过测试（绿）

这一步的要点是让测试通过，但没必要把实现做到完美。给`TicketRevenue`类一个更好的`estimateTotalRevenue`实现（不会只返回0），可以让测试通过（变绿）。

记住，这一阶段只要让测试通过就行，没必要追求完美。代码可能如代码清单11-2所示：

代码清单11-2　第一版通过测试的`TicketRevenue`

```
import java.math.BigDecimal;

public class TicketRevenue {

  public BigDecimal estimateTotalRevenue(int numberOfTicketsSold) {
    BigDecimal totalRevenue = BigDecimal.ZERO;
    if (numberOfTicketsSold == 1) {
      totalRevenue = new BigDecimal("30");        <-- 通过测试的实现
    }
    return totalRevenue;
  }
}
```

现在再运行测试，通过了！而且在大多数IDE中，会用一个绿条或对勾来表示测试通过。图11-1是在Eclipse中通过测试的界面。

接下来的问题是你能不能说“我搞定了”，然后去做下一项工作？我们可以负责任地告诉你：“不是！”你会忍不住想完善前面的代码，那现在我们就开始吧。

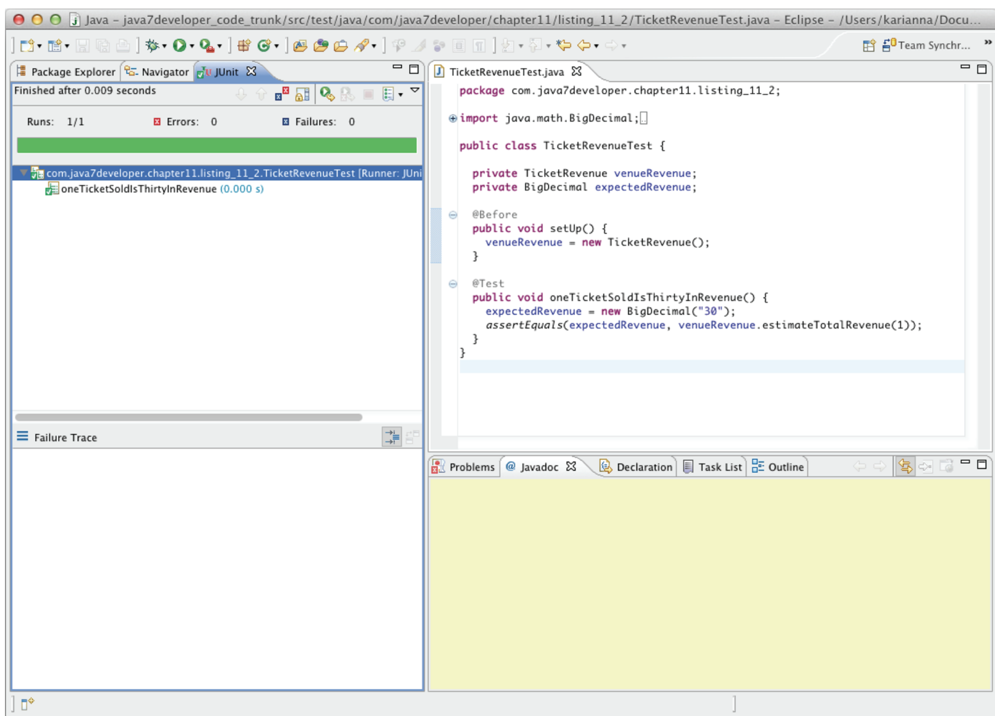


图11-1　Eclipse IDE中表示测试通过的绿条，纸质版印刷出来是中度灰色

3. 重构测试

这一步的要点是看看为了通过测试写的快速实现，确保你遵循了通行的惯例。代码清单11-2中的代码明显可以更清晰、更整洁。你肯定要重构，以减轻自己和他人的技术债务。

技术债务　Ward Cunningham发明的说法，指我们现在临时凑合出来的设计或代码将来会让我们付出更多的成本（工作）。

记住，有了通过测试，可以放心大胆地重构。应该实现的业务逻辑不可能会被忽视。

提示　编写最初的通过测试代码的另一个好处是开发进度可以更快。团队中的其他人可以马上用第一版代码跟更大的代码库一起测试（集成测试及更大范围的测试）。

在代码清单11-3中，我们不想再用魔法数字了——要让票价（30）出现在代码中。

代码清单11-3　通过测试的`TicketRevenue`重构版

```
import java.math.BigDecimal;
```

```
public class TicketRevenue {

  private final static int TICKET_PRICE = 30;          <-- 不用魔法数字了

  public BigDecimal estimateTotalRevenue(int numberOfTicketsSold) {
    BigDecimal totalRevenue = BigDecimal.ZERO;
    if (numberOfTicketsSold == 1) {
      totalRevenue =
        new BigDecimal(TICKET_PRICE *
                       numberOfTicketsSold);          <-- 重构的计算
    }
    return totalRevenue;
  }
}
```

经过这次重构，代码得到了改善，但很明显它还没有涵盖所有情况（售票数量为负值、0、2~100和大于100）。你不能只是拼命地猜其他情况下的实现应该是什么样，而应该做更多测试驱动的设计和实现。下一节会继续按照测试驱动设计的方式，带你看更多的测试用例。

11.1.2　多个测试用例

按照TDD风格，应该继续为门票销售数量为负值、0、2~100和大于100的情况依次添加测试用例。但还有一种办法，一次写一组测试用例也行，特别是在它们跟最初的测试有关的时候。

注意，这次仍然要遵循红—绿—重构的循环周期。在把这些用例都加上之后，你应该会得到一个带有失败测试（红）的测试类，如代码清单11-4所示。

代码清单11-4　`TicketRevenue`的失败单元测试

```
import java.math.BigDecimal;
import static junit.framework.Assert.*;
import org.junit.Test;

public class TicketRevenueTest {

  private TicketRevenue venueRevenue;
  private BigDecimal expectedRevenue;

  @Before
  public void setUp() {
    venueRevenue = new TicketRevenue();
  }

  @Test(expected=IllegalArgumentException.class)
  public void failIfLessThanZeroTicketsAreSold() {     <-- 销量为负值
    venueRevenue.estimateTotalRevenue(-1);
  }

  @Test
  public void zeroSalesEqualsZeroRevenue() {           <-- 销量为0
    assertEquals(BigDecimal.ZERO, venueRevenue.estimateTotalRevenue(0));
  }

  @Test
  public void oneTicketSoldIsThirtyInRevenue() {       <-- 销量为1
```

```
    expectedRevenue = new BigDecimal("30");
    assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(1));
  }

  @Test
  public void tenTicketsSoldIsThreeHundredInRevenue() {      <-- 销量为N
    expectedRevenue = new BigDecimal("300");
    assertEquals(expectedRevenue, venueRevenue.estimateTotalRevenue(10));
  }

  @Test(expected=IllegalArgumentException.class)
  public void failIfMoreThanOneHundredTicketsAreSold() {     <-- 销量大于100
    venueRevenue.estimateTotalRevenue(101);
  }
}
```

为通过所有测试（绿）写的基本实现版看起来应该如代码清单11-5所示。

代码清单11-5　通过测试的第一版`TicketRevenue`

```
import java.math.BigDecimal;

public class TicketRevenue {

  public BigDecimal estimateTotalRevenue(int numberOfTicketsSold)
    throws IllegalArgumentException {

    BigDecimal totalRevenue = null;
    if (numberOfTicketsSold < 0) {
      throw new IllegalArgumentException("Must be > -1");       <-- 异常情况
    }
    if (numberOfTicketsSold == 0) {
      totalRevenue = BigDecimal.ZERO;
    }
    if (numberOfTicketsSold == 1) {
      totalRevenue = new BigDecimal("30");
    }
    if (numberOfTicketsSold == 101) {
      throw new IllegalArgumentException("Must be < 101");      <-- 异常情况
    }
    else {
      totalRevenue =
          new BigDecimal(30 * numberOfTicketsSold);             <-- 销量为N
    }
    return totalRevenue;
  }
}
```

有了刚刚完成的实现，现在你的测试就变成通过测试了。

按照TDD循环周期，现在该重构这个实现了。比如说，可以把不合法的`numberOfTicketsSold`情况（负数或者大于100）放到一个`if`语句中，并用公式`(TICKET_PRICE * numberOfTicketsSold)`返回所有合法`numberOfTicketsSold`的收入。代码清单11-6应该跟重构之后的代码很像。

代码清单11-6　重构后的`TicketRevenue`版本

```
import java.math.BigDecimal;
```

```
public class TicketRevenue {
  private final static int TICKET_PRICE = 30;

  public BigDecimal estimateTotalRevenue(int numberOfTicketsSold)
    throws IllegalArgumentException {

    if (numberOfTicketsSold < 0 || numberOfTicketsSold > 100) {
      throw new IllegalArgumentException
                ("# Tix sold must == 1..100");          // 异常状况
    }

    return new BigDecimal
                (TICKET_PRICE * numberOfTicketsSold);   // 所有其他情况
  }
}
```

新的TicketRevenue类更加紧凑，并且还通过了所有测试！现在你已经完成了整个红—绿—重构循环，可以信心满满地开始实现下一个业务逻辑了。另外，如果你（或会计）发现漏掉了任何边界情况，比如有浮动票价，也可以再次开始一个循环。

我们强烈建议你弄明白红—绿—重构的TDD方式背后的原理，也就是我们接下来要讨论的内容。但如果你没什么耐心，可以直接跳到11.1.4节学习JUnit，或11.2节了解用来测试第三方代码的测试替身。

11.1.3　深入思考红—绿—重构循环

这一节会在前面例子的基础上探索TDD背后的一些思想。我们会再次谈论红—绿—重构循环，你应该还记得第一步是写失败测试。但这也有几种不同的方式。

1. 失败测试（红）

一些开发人员真的喜欢编写编译失败的测试，喜欢等到绿色步骤才提供实现代码。也有一些开发人员喜欢先把测试调用的方法存根写出来，这样虽然测试代码能编译，但还是会失败。我们觉得怎么样都行，随意就好。

提示　这些测试代码是实现的第一个客户，所以应该认真考虑该怎么设计它们：方法定义看起来应该是什么样的。还应该问自己几个问题：该传什么参数进去？期望的返回值是什么？会不会有异常情况？另外，不要忘了测试重要领域对象的equals()和hashCode()方法。

一旦写完失败测试，就该进入下一阶段了：让它通过。

2. 通过测试（绿）

这一步应该尽量少写代码，只要保证测试通过就行。也就是说你不用把实现做到完美，那是重构阶段的工作。

测试通过之后，你就可以告诉同事，你的代码已经实现了它应该实现的功能，他们可以拿去用了。

3. 重构

在这一步中应该重构实现代码。可以重构的地方数不胜数，但有几个应该重点关注的，比如

去掉硬编码的变量或把大方法拆分开。如果是面向对象的代码，则应该遵循SOLID原则。

SOLID原则是Bob大叔（Robert Martin）提出来的，请参见表11-2。要了解更详细的信息，可以参考他的文章“The Principles of OOD”（http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod）。

表11-2 面向对象代码的SOLID原则

原 则	描 述
单一职责原则（SRP）	每个对象都应该做一件事，并且只做一件事
开放/封闭原则（OCP）	对象应该是可扩展、但不可修改的
里氏替换原则（LSP）	对象应该可以被它的子类型实例替换
接口隔离原则（ISP）	特定的小接口更好
依赖倒置原则（DIP）	不要依赖具体实现（请参见第3章关于依赖注入的内容）

提示 我们还要向你推荐Checkstyle和FindBugs这两个静态代码分析工具（第12章还有更多）。Joshua Bloch的*Effective Java, Sencond Edition*[①]（Addison-Wesley, 2008）也是好资源，其中有很多Java语言的技巧和窍门。

测试代码本身的重构是个容易被人遗忘的角落。你可以把通用的设置和拆卸代码提取出来，可以重命名测试以更准确地反应它的测试意图，还可以根据静态分析工具的分析结果做些小修订。

现在你已经能跟上TDD的三步走了，该去熟悉一下JUnit了，它可是在Java里写TDD代码的默认工具。

11.1.4 JUnit

JUnit是公认的Java项目测试框架。当然，除了JUnit还有其他测试框架，比如拥有不少追随者的TestNG，但目前JUnit还是Java测试界的主流。

注意 如果你熟悉JUnit，可以跳到11.2节。

JUnit有三个主要特性：

- 用于测试预期结果和异常的断言，比如`assertEquals()`；
- 设置和拆卸通用测试数据的能力，比如`@Before`和`@After`；
- 运行测试套件的测试运行器。

JUnit用简单的注解模型提供了很多重要的功能。

大多数IDE（比如Eclipse、IntelliJ和NetBeans）都内置了JUnit，如果你用的正好是其中之一，就不用自己去下载、安装或配置JUnit了。如果你的IDE没有安装JUnit，可以访问www.junit.org查

① 《Effective Java中文版》由机械工业出版社于2003年出版。——编者注

看它的下载和安装指导[①]。

> **注意** 我们用的是JUnit 4.8.2。如果你要练习本章中的例子，建议也用这个版本。

一个基本的JUnit测试包含下面这些元素：

- 用`@Before`标记设置方法，在每个测试运行前准备测试数据；
- 用`@After`标记拆卸方法，在每个测试运行完成后拆卸测试数据；
- 测试方法本身（用`@Test`注解标记）。

为了多了解一下上面这些元素，我们来看几个非常基本的JUnit测试。

比如OpenJDK团队要你给`BigDecimal`类写个单元测试。第一个测试是检查加法`(1.5 + 1.5==3.0);`，第二个测试是检查用非数字值创建`BigDecimal`实例时会抛出`NumberFormatException`异常。

> **注意** 我们在本章的例子中经常同时给出多个失败测试，实现（绿）和重构。这违背了纯粹的TDD单个测试贯穿红—绿—重构循环的原则，但却可以让我们在本章中放入更多例子。不过在你编码时，应该尽可能地遵守单个测试循环的开发模型。

要运行代码清单11-7，可以在IDE里的源码文件上点击右键，选择运行或测试选项（三个主流IDE中都有显眼的Run Test或Run File选项）。

代码清单11-7 JUnit测试的基本结构

```java
import java.math.BigDecimal;
import org.junit.*;
import static org.junit.Assert.*;

public class BigDecimalTest {

  private BigDecimal x;

  @Before
  public void setUp() { x = new BigDecimal("1.5"); }

  @After
  public void tearDown() { x = null; }

  @Test
  public void addingTwoBigDecimals() {
    assertEquals(new BigDecimal("3.0"), x.add(x));
  }

  @Test(expected=NumberFormatException.class)
  public void numberFormatExceptionIfNotANumber() {
    x = new BigDecimal("Not a number");
  }

}
```

标准的**JUnit**导入

❶ 每个测试之前的设置

❷ 每个测试之后的拆卸

❸ 执行测试

❹ 处理意料中的异常

① 第12章会讲到JUnit和Maven的集成。

在每个测试运行之前，x在@Before区域中被设置为BigDecimal("1.5")❶。这会确保每个测试处理的都是已知值x，而不是被之前运行的测试修改过的中间值。在每个测试运行之后，在@After区域中确保x被设为null❷（以便x可以被垃圾收集）。然后用assertEquals()（JUnit众多静态assertX方法之一）测试BigDecimal.add()的返回结果是否符合期望❸。为了处理预期的异常，在@Test上加上了可选的expected参数❹。

进入TDD最佳状态的最好办法就是动手实践。把TDD原则牢牢印在你的脑海里，把JUnit框架搞明白，你就可以开始了！通过这些例子你也能看出来，单元测试级的TDD很容易掌握。

但所有TDD从业者最终都要测试使用依赖项或子系统的代码。下一节就会讲到那些代码的测试技术。

11.2 测试替身

如果你继续用TDD风格编码，很快就会遇到需要引用（经常是第三方的）依赖项或子系统的情况。在这种情况下，你肯定想把测试代码跟依赖项隔离开，以保证测试代码仅仅针对于实际构建的代码。你肯定还想让测试代码尽可能快速运行。而调用第三方依赖项或子系统（比如数据库）可能会花很长时间，也就是说会丧失TDD快速响应的优势（在单元测试层面尤其如此）。测试替身（test double）就是为解决这个问题而生的。

你在这一节将学会如何用测试替身有效隔离依赖项和子系统，看到使用四种测试替身（虚设、伪装、存根和模拟）的例子。

在最复杂的情况下，也就是测试有外部依赖项（比如分布式服务或网络服务）的代码时，依赖注入技术（见第3章）会和测试替身联手来拯救你，即便是看上去大得吓人的系统，它们也能保你安全无虞。

为什么不用Guice？

如果对第3章还记忆犹新，你应该不会忘了Guice——Java DI框架的参考实现。阅读这一节时你很可能边看边想："他们怎么不用Guice呢？"

简言之，对于这些代码，即便引入像Guice这样简单的框架都显得过于复杂。记住，DI是一项技术。不要纯粹为了使用框架而使用它。

Gerard Meszaros在他的*xUnit Test Patterns*[①]（Addison-Wesley Professional，2007）一书中给出了测试替身的简单解释，我们很荣幸能在这里引用他的说法："测试替身（想一想特技演员）泛指任何出于测试目的替换真实对象的假冒对象。"

Meszaros接着定义了四种测试替身，如表11-3所示。

虽然看起来很抽象，但见到例子你就知道了，它们非常容易理解。让我们先从虚设对象开始讲起。

① 本书中文版《xUnit测试模式：测试码重构》已由清华大学出版社于2009年出版。——译者注

表11-3　四种测试替身

类　型	描　述
虚设替身	只传递不使用的对象。一般用于填充方法的参数列表
存根替身	总是返回相同预设响应的对象，其中可能也有些虚设状态
伪装替身	可以取代真实版本的可用版本（当然在品质和配置上达不到生产环境要求的标准）
模拟替身	可以表示一系列期望值的对象，并且可以提供预设响应

11.2.1　虚设对象

在这四种测试替身里，虚设对象用起来最容易。记住，它是用来填充参数列表，或者填补那些总也不会用的必填域。大多数情况下，你甚至可以传入一个空对象或`null`。

我们回到剧院门票那个例子中。能估算出一个售票亭带来的收入非常好，但剧院老板考虑得更长远。售出门票和预期收入的模型要做得更好，并且你还听到有人抱怨：随着需求增多，系统越来越复杂了。

你接到一项任务，要对售出票进行跟踪，并且某些票可以打9折。看起来你需要一个带有价格打折方法的`Ticket`类。你又从TDD循环的失败测试开始了，测试重点是新的`getDiscountPrice()`方法。你知道还需要两个构造方法：一个用于常规价格的门票，一个用于可能会打折的门票。`Ticket`对象最终需要两个参数：

- 客户姓名，测试中绝不会用到的`String`；
- 正常价格，测试中会用到的`BigDecimal`。

你非常确定`getDiscountPrice()`方法肯定不会引用客户姓名，也就是说可以给构造方法传入一个虚设对象（我们用的是固定字符串`"Riley"`），如代码清单11-8所示。

代码清单11-8　用虚设对象实现的`TicketTest`

```
import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;

public class TicketTest {

  @Test
  public void tenPercentDiscount() {
    String dummyName = "Riley";                          ◁ 创建虚设对象
    Ticket ticket = new Ticket(dummyName,
                               new
     BigDecimal("10"));                                  ◁ 传入虚设对象
    assertEquals(new BigDecimal("9.0"), ticket.getDiscountPrice());
  }

}
```

看到了吧，虚设对象的概念很平常。

为了让你彻底明白这个概念，我们在代码清单11-9中给出了部分实现的`Ticket`类。

代码清单11-9 用虚设对象测试Ticket类

```
import java.math.BigDecimal;

public class Ticket {
  public static final int BASIC_TICKET_PRICE = 30;          ← 默认价格
  private static final BigDecimal DISCOUNT_RATE =
                              new BigDecimal("0.9");        ← 默认折扣

  private final BigDecimal price;
  private final String clientName;

  public Ticket(String clientName) {
    this.clientName = clientName;
    price = new BigDecimal(BASIC_TICKET_PRICE);
  }

  public Ticket(String clientName, BigDecimal price) {
    this.clientName = clientName;
    this.price = price;
  }

  public BigDecimal getPrice() {
    return price;
  }

  public BigDecimal getDiscountPrice() {
    return price.multiply(DISCOUNT_RATE);
  }
}
```

有些开发人员会被虚设对象搞糊涂——他们预期的复杂度并不存在。虚设对象非常直接，它们就是过去为了避免出现NullPointerException的古老对象，只是为了让代码能跑起来。

我们转入下一个测试替身的讨论吧。存根对象（从复杂度来讲）向前迈出了一步。

11.2.2 存根对象

在使用能够做出相同响应的对象代替真实实现的情况下，就会用到存根对象。让我们回到剧院门票价格的例子中，看一下实际应用。

写完Ticket类后，领导给你放了个假。你度完假刚回来，打开邮箱就看到一个bug单，报告说代码清单11-8中的tenPercentDiscount()测试时好时坏。你一检查代码库，发现tenPercentDiscount()已经被改掉了。现在新写了一个Price接口，而Ticket实例是由该接口的实现类HttpPrice创建的。

经过调查，你又发现一些变化，为了从一个外部网站上的第三方类HttpPricingService获得最初的价格，要调用HttpPrice的getInitialPrice()方法。

因此每次调用getInitialPrice()都会返回不同的价格。此外，它时好时坏还有几个原因，有时是公司防火墙规则变了，有时是第三方网站无法访问了。

所以测试就失败了，测试的目的也不幸受到了污染。记住，你所要的单元测试只是针对打9折的价格。

注意　涉及第三方价格网站调用的情景肯定超出了测试的责任范围。但你可以考虑做一个单独覆盖HttpPrice类和第三方的HttpPricingService的系统集成测试。

在用存根替换HttpPrice类之前，先看一下代码的当前状态，如下面三段代码（代码清单11-10至代码清单11-12）。除了跟Price接口有关的修改，剧院老板的想法也变了，觉得没必要记录是谁买了票，代码如下所示。

代码清单11-10　实现了新需求的TicketTest

```
import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;

public class TicketTest {

  @Test
  public void tenPercentDiscount() {
    Price price = new HttpPrice();                    // 实现了Price的HttpPrice
    Ticket ticket = new Ticket(price);                // 创建Ticket
    assertEquals(new BigDecimal("9.0"),
                 ticket.getDiscountPrice());           // 测试可能会失败
  }
}
```

下面是新的Ticket，现在它包括了一个私有类FixedPrice，用来处理价格已知并固定的情况，即不需要从外部源中获取这些信息。

代码清单11-11　实现了新需求的Ticket

```
import java.math.BigDecimal;

public class Ticket {
  public static final int BASIC_TICKET_PRICE = 30;
  private final Price priceSource;
  private BigDecimal faceValue = null;
  private final BigDecimal discountRate;

  private final class FixedPrice implements Price {
    public BigDecimal getInitialPrice() {
      return new BigDecimal(BASIC_TICKET_PRICE);
    }
  }

  public Ticket() {                                   // 修改过的构造方法
    priceSource = new FixedPrice();
    discountRate = new BigDecimal("1.0");
  }

  public Ticket(Price price) {                        // 修改过的构造方法
    priceSource = price;

   discountRate = new BigDecimal("1.0");
  }
```

```
  public Ticket(Price price,
                BigDecimal specialDiscountRate) {        <-- 修改过的构造方法
    priceSource = price;
    discountRate = specialDiscountRate;
  }

  public BigDecimal getDiscountPrice() {
    if (faceValue == null) {
     faceValue = priceSource.getInitialPrice();          <-- 新的getInitialPrice方法调用
    }
    return faceValue.multiply(discountRate);             <-- 计算没变化
  }
}
```

代码清单11-12 Price接口及其实现HttpPrice

```
import java.math.BigDecimal;

public interface Price {
  BigDecimal getInitialPrice();
}

public class HttpPrice implements Price {
  @Override
  public BigDecimal getInitialPrice() {
    return HttpPricingService.getInitialPrice();         <-- 返回结果随机
  }
}
```

那么，怎么才能做出跟HttpPricingService一样的响应？关键是想清楚测试的真实意图是什么？在这个例子中，你要测的是Ticket类中getDiscountPrice()方法所做的乘法跟预期一致。

因此你可以用总是返回同一价格的存根StubPrice换掉HttpPrice类，以调用getInitialPrice()。这样就可以把价格经常变化且时好时坏的HttpPrice类从测试中隔离出去了。使用代码清单11-13中的实现，测试就可以通过了。

代码清单11-13 使用存根对象的TicketTest实现

```
import org.junit.Test;
import java.math.BigDecimal;
import static org.junit.Assert.*;

public class TicketTest {

  @Test
  public void tenPercentDiscount() {
    Price price = new StubPrice();                       <-- StubPrice存根
    Ticket ticket = new Ticket(price);                   <-- 创建Ticket
    assertEquals(9.0,
                 ticket.getDiscountPrice().doubleValue(),
                 0.0001);                                <-- 检查价格
  }
}
```

StubPrice是个简单的小类，返回的初始价格总是10，如代码清单11-14所示。

代码清单11-14　存根StubPrice

```
import java.math.BigDecimal;

public class StubPrice implements Price {

  @Override
  public BigDecimal getInitialPrice() {
    return new BigDecimal("10");          <-- 返回同一价格
  }
}
```

咻！现在测试又能通过了，重要的是你又可以毫不畏惧地重构剩下的实现细节了。

存根是种挺实用的测试替身，但有时候我们会希望存根所做的工作可以尽可能地接近生产系统，这时可以用伪装替身。

11.2.3　伪装替身

伪装对象可以看做是存根的升级，它所做的工作几乎和生产代码一样，但为了满足测试需求会走些捷径。如果你想让代码的运行时环境非常接近生产环境（连接真实的第三方子系统或依赖项），伪装替身特别有用。

大部分Java开发人员迟早都要编写跟数据库交互的代码，特别是在Java对象上执行CRUD操作。在DAO（Data Access Object，数据访问对象）代码跟生产数据库连接之前，证明其可用的工作通常会留到系统集成测试阶段，或者根本就不做检查！如果能在单元测试或集成测试阶段对DAO代码进行检查，那将会有很多好处，最重要的是你能快速响应。

在这种情况下可以用伪装对象：用来代表跟你交互的数据库。但自己写一个代表数据库的伪装对象相当困难！好在经过数年的演进，内存数据库的轻巧易用已经足以胜任这一工作。HSQLDB（www.hsqldb.org）是广泛用于这一用途的内存数据库。

剧院门票应用进展良好，下一阶段的工作就是把门票保存在数据库中，以便后期获取。Java中最常用的数据库持久化框架是Hibernate（www.hibernate.org）。

Hibernate与HSQLDB

如果你不了解Hibernate或HSQLDB，请不要惊慌！Hibernate是一个对象关系映射（ORM）框架，实现了Java持久化API（JPA）标准。简而言之，你可以调用简单的`save`、`load`、`update`，还有很多其他的Java方法来执行CRUD操作。这和用原始的SQL和JDBC不同，并且它经过抽象隔离了特定数据库的语法和语义。

HSQLDB只是个Java内存数据库。只要把hsqldb.jar放到你的CLASSPATH下就可以用了。尽管在关闭之后数据会全部丢失，但它的表现跟一般的RDBMS很像。（其实数据是可以保存下来的，请访问HSQLDB的网站了解更多细节。）

虽然我们可能又扔给你两项新技术，但随书源码中的构建脚本会帮你把正确的JAR依赖项和配置文件放到正确的地方。

首先，你需要一个Hibernate配置文件来定义到HSQLDB数据库的连接，如代码清单11-15所示。

代码清单11-15 用于HSQLDB的Hibernate配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">
      org.hibernate.dialect.HSQLDialect                    设置方言
    </property>
    <property name="hibernate.connection.driver_class">
      org.hsqldb.jdbcDriver
     </property>
    <property name="hibernate.connection.url">
      jdbc:hsqldb:mem:wgjd                                 指定要连接的URL
    </property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.autocommit">true</property>
    <property name="hibernate.hbm2ddl.auto">
      create
    </property>                                            自动创建数据表
    <property name="hibernate.show_sql">true</property>
    <mapping resource="Ticket.hbm.xml"/>                   ❶ 映射Ticket类
  </session-factory>
</hibernate-configuration>
```

你应该注意到了，清单中的最后一行语句引用了Ticket类的映射资源（<mapping resource="Ticket.hbm.xml"/>）❶。这个资源会告诉Hibernate怎么把Java文件映射到数据库列。在Hibernate配置文件里，除了方言（HSQLDB），还有所有Hibernate需要用来在幕后自动构建SQL的信息。

尽管Hibernate允许你在Java类里直接用注解添加映射信息，但我们还是更喜欢下面这种XML映射方式，如代码清单11-16所示。

> **警告** 注解跟XML映射之间的选择之战在邮件列表中已经打了很久了，所以你最好选个自己喜欢的，然后就由它去吧。

11

代码清单11-16 用于Ticket的Hibernate映射文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

  <class
   name="com.java7developer.chapter11
➥ .listing_11_18.Ticket">                                 标出要映射的类
```

```
    <id name="ticketId"
        type="long"
        column="ID" />                                        指定ticketId为关键字

    <property name="faceValue"
              type="java.math.BigDecimal"
              column="FACE_VALUE"
              not-null="false" />                              faceValue映射

    <property name="discountRate"
              type="java.math.BigDecimal"
              column="DISCOUNT_RATE"
              not-null="true" />                               discountRate映射

  </class>
</hibernate-mapping>
```

弄完配置文件，该想想测什么了。用唯一ID获取Ticket是业务需要。为了满足这一业务（和Hibernate映射）要求，必须将Ticket类改成代码清单11-17这样。

代码清单11-17　带有ID的Ticket

```
import java.math.BigDecimal;

public class Ticket {

  public static final int BASIC_TICKET_PRICE = 30;
  private long ticketId;                                  <-- 加上ID
  private final Price priceSource;
  private BigDecimal faceValue = null;
  private BigDecimal discountRate;

  private final class FixedPrice implements Price {
    public BigDecimal getInitialPrice() {
      return new BigDecimal(BASIC_TICKET_PRICE);
    }
  }

  public Ticket(long id) {
    ticketId = id;
    priceSource = new FixedPrice();
    discountRate = new BigDecimal("1.0");
  }

  public void setTicketId(long ticketId) {
    this.ticketId = ticketId;
  }

  public long getTicketId() {
    return ticketId;
  }

  public void setFaceValue(BigDecimal faceValue) {
    this.faceValue = faceValue;
  }

  public BigDecimal getFaceValue() {
    return faceValue;
```

```
  }

  public void setDiscountRate(BigDecimal discountRate) {
    this.discountRate = discountRate;
  }

  public BigDecimal getDiscountRate() {
    return discountRate;
  }

  public BigDecimal getDiscountPrice() {
    if (faceValue == null) faceValue = priceSource.getInitialPrice();
    return faceValue.multiply(discountRate);
  }
}
```

现在Ticket的映射有了，Ticket类也改过了，可以调用TicketHibernateDao里的findTicketById方法进行测试了。哦，还要写JUnit测试设置的准备代码，如代码清单11-18所示：

代码清单11-18　TicketHibernateDaoTest测试类

```
import java.math.BigDecimal;
import org.hibernate.cfg.Configuration;
import org.hibernate.SessionFactory;
import org.junit.*;
import static org.junit.Assert.*;

public class TicketHibernateDaoTest {

  private static SessionFactory factory;
  private static TicketHibernateDao ticketDao;
  private Ticket ticket;
  private Ticket ticket2;

  @BeforeClass
  public static void baseSetUp() {
    factory =
      new Configuration().
          configure().buildSessionFactory();
    ticketDao = new TicketHibernateDao(factory);          ❶ 使用Hibernate配置
  }

  @Before
  public void setUpTest()
  {
    ticket = new Ticket(1);
    ticketDao.save(ticket);
    ticket2 = new Ticket(2);                              ❷ 设置测试Ticket的数据
    ticketDao.save(ticket2);
  }

  @Test
  public void findTicketByIdHappyPath() throws Exception {
    Ticket ticket = ticketDao.findTicketById(1);
    assertEquals(new BigDecimal("30.0"),                  ❸ 找到Ticket
                 ticket.getDiscountPrice());
  }
```

```
  @After
  public static void tearDown() {
    ticketDao.delete(ticket);                          清除数据
    ticketDao.delete(ticket2);
  }

  @AfterClass
  public static void baseTearDown() {                   关闭
    factory.close();
  }

}
```

在运行任何测试之前，先用Hibernate的配置创建所要测试的DAO❶。然后，在每个测试运行之前，都在HSQLDB数据库里存两条门票的记录（作为测试数据）❷。运行测试，测试DAO的`findTicketById`方法❸。

因为你还没写`TicketHibernateDao`类及其方法，所以测试一开始会失败。使用Hibernate框架不需要SQL，也不需要提及用的是HSQLDB数据库。因此，DAO的实现应该和代码清单11-19类似。

代码清单11-19　`TicketHibernateDao`类

```
import java.util.List;
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.criterion.Restrictions;

public class TicketHibernateDao {

  private static SessionFactory factory;
  private static Session session;

  public TicketHibernateDao(SessionFactory factory)
  {
    TicketHibernateDao.factory = factory;              设置工厂和会话
    TicketHibernateDao.session = getSession();
  }

  public void save(Ticket ticket)
  {
    session.save(ticket);                              ❶ 保存Ticket
    session.flush();
  }

  public Ticket findTicketById(long ticketId)
  {
    Criteria criteria =
        session.createCriteria(Ticket.class);
    criteria.add(Restrictions.eq("ticketId", ticketId));   ❷ 使用ID查找Ticket
    List<Ticket> tickets = criteria.list();
    return tickets.get(0);
  }
```

```
  public void delete(Ticket ticket) {
    session.delete(ticket);
    session.flush();
  }

  private static synchronized Session getSession() {
    return factory.openSession();
  }

}
```

DAO的save方法特别不起眼，就是调用Hibernate的save方法，然后用flush确保对象能存到HSQLDB数据库中❶。要取出Ticket，可以用Hibernate的Criteria（相当于SQL里的WHERE从句）❷。

写完DAO之后，测试就能通过了。你可能已经注意到了，save方法也已经被部分测试到了。你可以继续写更加完整的测试，比如检查一下从数据库中取回的票是否带有正确的discount-Rate。现在可以提前测试数据库访问代码了，所以数据库访问层也得到了TDD方式的所有好处。

我们接着讨论下一个测试替身：模拟对象。

11.2.4 模拟对象

模拟对象跟前面提过的存根对象是亲戚，但存根对象一般都特别呆。比如在调用存根时它们通常总是返回相同的结果。所以不能模拟任何与状态相关的行为。

看个例子：假设你想用TDD方式写一个文本分析系统。其中一个单元测试要求文本分析类对某篇博文中出现的“Java 7”进行计数。但这篇博文是第三方资源，所以很多失败都跟你写的计数算法没太大关系。换句话说，测试代码不是孤立的，并且获取第三方资源可能很费时间。下面是一些很常见的失败：

- 由于防火墙限制，你的代码可能无法访问互联网上的这篇博文；
- 这篇博文可能被挪走了，而链接没有重定向；
- 博文可能被编辑过，“Java 7”出现的次数可能增加了，也可能减少了。

用存根几乎不可能把这个测试写出来，即便能写也极其繁琐，模拟对象此时登场。这是一种特殊的测试替身，你可以把它当做可以预编程的存根或超级存根。使用模拟对象非常简单：在准备要用的模拟对象时，告诉它预计会有哪些调用，以及每个调用该如何响应。模拟会跟DI结合得很好，你可以用它注入一个虚拟的对象，这个对象将完全按照已知方式行动。

让我们看一个剧院门票的例子。我们会用一个流行的模拟类库Mockito（http://mockito.org/），请看代码清单11-20。

代码清单11-20 用于剧院门票的模拟对象

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

import java.math.BigDecimal;
import org.junit.Test;
```

```
public class TicketTest {

  @Test
  public void tenPercentDiscount() {
    Price price = mock(Price.class);                    ❶ 创建模拟对象
    when(price.getInitialPrice()).
        ➥ thenReturn(new BigDecimal("10"));            ❷ 对模拟对象编程以便进行测试

    Ticket ticket = new Ticket(price, new BigDecimal("0.9"));
    assertEquals(9.0, ticket.getDiscountPrice().doubleValue(), 0.000001);

    verify(price).getInitialPrice();
  }
}
```

创建模拟对象需要调用静态的`mock()`方法❶，并将模拟目标类型的class对象作为参数传给它。然后要把模拟对象需要表现出来的行为记录下来，通过调用`when()`方法表明要记录哪些方法的行为，然后用`thenReturn()`指定所期望的结果是什么❷。最后要证实在模拟对象上调用了预期的方法。这是为了确保你的正确结果不是经由不正确的路径得到的。

你可以像使用常规对象那样使用模拟对象，并且无需任何其他步骤就可以把它传给你调用的`Ticket`构造方法。这使得模拟对象成为了TDD的得力工具，有些从业者实际上更喜欢所有事情都用模拟对象来做，完全放弃了其他测试替身。

不管你是不是选择这种“最模拟”的TDD风格，完整的测试替身（需要的话加上一点DI）知识会让你毫不畏惧地进行重构和编码，即便面对复杂的依赖和第三方子系统也不怕。

Java开发人员会发现TDD的工作方式非常容易上手。但Java经常伴随着一个反复出现的问题——有些繁琐。在纯粹的Java项目中用TDD会导致大量的套路化代码。好在现在你已经学了一些其他的JVM语言，能用它们做出更精炼的TDD。实际上，从测试开始将非Java语言带入项目中是推动多语言项目的经典方式之一。

在下一节中，我们会讨论ScalaTest，这个测试框架具有广泛的测试用途。我们会从介绍ScalaTest开始，并会向你展示如何用它运行JUnit测试来测试Java类。

11.3　ScalaTest

如果你还记得，我们在7.4节说过TDD是动态语言的理想用例。实际上，Scala先进的类型推断让它在做测试上同样也有很多优势，尽管它是静态类型系统，还是经常会让人觉得它是动态语言。

Scala中的主测试框架是ScalaTest。它为做各种测试提供了一些极其实用的特质和类——从JUnit风格的单元测试到全面的集成和验收测试。我们来看一个ScalaTest的实战例子。

代码清单11-21用ScalaTest重写了11.4节中的代码，并且加了一个新的`sellTicket()`方法测试`fiftyDiscountTickets()`。

代码清单11-21 ScalaTest风格的JUnit测试

```
import java.math.BigDecimal
import java.lang.IllegalArgumentException
import org.scalatest.junit.JUnitSuite
import org.scalatest.junit.ShouldMatchersForJUnit
import org.junit.Test
import org.junit.Before
import org.junit.Assert._

class RevenueTest extends JUnitSuite with ShouldMatchersForJUnit {

  var venueRevenue: TicketRevenue = _

  @Before def initialize() {
    venueRevenue = new TicketRevenue()
  }

  @Test def zeroSalesEqualsZeroRevenue() {
    assertEquals(BigDecimal.ZERO, venueRevenue estimateTotalRevenue 0);
  }

  @Test def failIfTooManyOrTooFewTicketsAreSold() {
    evaluating { venueRevenue.estimateTotalRevenue(-1) }
➥ should produce [IllegalArgumentException]              <- 预期的异常
    evaluating { venueRevenue.estimateTotalRevenue(101) }
➥ should produce [IllegalArgumentException]
  }

  @Test def tenTicketsSoldIsThreeHundredInRevenue() {
    val expected = new BigDecimal("300");
    assert(expected == venueRevenue.estimateTotalRevenue(10));
  }

  @Test def fiftyDiscountTickets() {
    for (i <- 1 to 50)
➥ venueRevenue.sellTicket(new Ticket())
    for (i <- 1 to 50)
➥ venueRevenue.sellTicket(new Ticket(new StubPrice(),
➥ new BigDecimal(0.9)))
    assert(1950.0 ==
➥ venueRevenue.getRevenue().doubleValue());      <- Scala风格的断言
  }
}
```

我们还没讲过Scala如何处理注解。它们看起来跟Java注解一样。这没什么好说的。你的测试也是放在扩展了`JUnitSuit`的类中，这就是说ScalaTest会把这个类当做它能运行的东西。

你可以在命令行中用本地ScalaTest运行器轻松运行ScalaTest：

```
ariel:scalatest boxcat$ scala -cp /Users/boxcat/projects/tickets.jar:/Users/
     boxcat/projects/wgjd/code/lib/scalatest-1.6.1.jar:/Users/boxcat/
     projects/wgjd/code/lib/junit-4.8.2.jar org.scalatest.tools.Runner -o -s
     com.java7developer.chapter11.scalatest.RevenueTest
```

在这条命令中，所测试的Java类放在tickets.jar文件中，所以要把它跟ScalaTest和JUnit文件一起放在类路径中。

这条命令用-s选项指定了要运行的测试集（省略-s选项会运行所有测试集中的所有测试）。-o选项把测试输出发送到标准输出中（用-e把测试结果输出到标准错误流中）。ScalaTest参照这个配置输出报道途径（包括其他途径，比如图形化界面）。前面的例子产生的输出如下所示：

```
Run starting. Expected test count is: 4
RevenueTest:
- zeroSalesEqualsZeroRevenue
- failIfTooManyOrTooFewTicketsAreSold
- tenTicketsSoldIsThreeHundredInRevenue
- fiftyDiscountTickets
Run completed in 820 milliseconds.
Total number of tests run: 4
Suites: completed 1, aborted 0
Tests: succeeded 4, failed 0, ignored 0, pending 0
All tests passed.
```

这些测试已经被编译进了一个类文件中。只要类路径中有JUnit和ScalaTest两者的JAR，就可以用scala运行这些测试，而不用在JUnit运行器中。

```
ariel:scalatest boxcat$ scala -cp /Users/boxcat/projects/tickets.jar:/Users/
     boxcat/projects/wgjd/code/lib/scalatest-1.6.1.jar:/Users/boxcat/
     projects/wgjd/code/lib/junit-4.8.2.jar org.junit.runner.JUnitCore
     com.java7developer.chapter11.scalatest.RevenueTest
JUnit version 4.8.2
...
Time: 0.096

OK (4 tests)
```

当然，输出会稍有不同，因为执行测试用的是不同的工具（JUnit运行器）。

注意　在用Maven构建第12章的java7developer项目时，我们会用这个JUnit运行器。

用ScalaTest测试Scala代码

我们在这一节主要讨论用ScalaTest测试Java代码。但如果你用Scala作为项目中的主要编程语言会怎么样?

人们通常认为Scala是稳定层语言，所以如果你在使用Scala代码，应该也可以像测试Java代码那样测试Scala代码库。所以用ScalaTest代替JUnit是使用TDD方式的不二之选。

快速了解ScalaTest后，我们对TDD的讨论就结束了。

11.4　小结

测试驱动开发能消除或减轻开发过程中的恐惧。遵从TDD风格，比如单元测试的红—绿—重构循环，开发人员可以把自己从思维定式中解放出来，不会步入临时拼凑代码的窘境。

JUnit是Java开发人员的主要测试类库。它可以指定设置和拆卸挂钩，运行一个测试集里相互独立的测试。JUnit的断言机制会判断调用实现逻辑后是否能产生想要的结果。

不同类型的测试替身可以帮你写出恰当的测试。你可以用四种测试替身（虚设、存根、伪装和模拟）取代依赖项，从而让测试精准运行。在编写测试代码时，借助模拟对象可以实现终极的灵活性。

ScalaTest始终秉持大量减少套路化测试代码的观念，有助于开发人员深入理解测试的行为驱动开发风格。

我们在下一章讨论自动构建，以及建立在TDD基础之上的持续集成（CI）开发方法。使用CI开发方法，你能立即得到每个新变化的自动反馈，并且它鼓励开发团队成员之间彻底透明化。

第 12 章

构建和持续集成

本章内容

- ❑ 构建管道和持续集成（CI）的重要性
- ❑ Maven 3：惯例优先于配置的构建工具
- ❑ Jenkins：得到公认的CI工具
- ❑ 使用FindBugs和Checkstyle等静态代码分析工具
- ❑ Leiningen：Clojure构建工具

我们接下来要讲的故事取材于MegaCorp的真实事件，出于对当事人的保护隐去了真实姓名。故事的主角是：

- ❑ Riley，刚毕业的新人；
- ❑ Alice和Bob，两个“经验丰富”的开发老手；
- ❑ Hazel，紧张的项目经理。

时间是周五下午两点，新开发的Sally支付功能要在周末跑批前上线。

Riley：我能为上线做点什么吗?

Alice：当然，我想最后一版是Bob构建的。Bob?

Bob：是的，是我几周前用Eclipse生成的。

Riley：但现在我们都用IntelliJ了；那么，该怎么构建呢?

Bob：哦，这需要些经验！总之我们会搞定它，年轻人!

Riley：好。我没这方面的经验，但支付功能的构建应该没问题，对吧?

Alice：当然没问题。我在两周前刚创建的代码分支，其他人对代码的改动肯定还不多。

Bob：但是，实际上，你知道我们添了些泛型的修改，对不对?

[尴尬的沉默]

Hazel：改完你们要经常在一起试试。这个我们强调过很多次了！

Riley：要不要我订外卖？貌似今晚我们得加班了。

Hazel：你说对了，学得挺快嘛!

Alice：实际上，我已经将订餐电话设成快速拨号状态了，这是常态!

Hazel：赶紧把它搞定！我们已经因为延迟发布和bug太多损失很多了，高管正想找机会杀鸡儆猴呢。

Alice、Bob和Riley明显没有优秀的构建和持续集成（CI）经验，但“构建和CI”究竟是什么意思？

构建和持续集成　快速和重复地为各种环境产生高质量二进制部署工件的过程。

开发团队经常谈论“构建”或“构建过程”[①]。就本章而言，我们在提到构建时是指遵循构建周期用构建工具将源码转化成二进制工件的过程。像Maven这种构建工具有很长的、详细的构建周期，它们中的大多数对于开发人员来说是不可见的。一个相当基础的、典型的构建周期如图12-1所示。

清除 ——→ 编译 ——→ 测试 ——→ 打包

图12-1　一个简化的典型构建周期

持续集成是指团队成员按照“尽早提交，经常提交”的口号频繁地集成工作成果。每个开发人员至少按天把代码提交到版本控制系统中，CI服务器会自动定期构建，以尽快检查集成错误[②]。CI服务器通常会在大屏幕上显示开心/悲伤的表情给团队以反馈。

那么构建和CI为什么重要？本章的每一节都会强调某些独特的好处，表12-1中列出了其中最为重要的几个。

表12-1　构建和CI的主要优势

主　　题	解　　释
重复性	任何人都可以随时随地运行构建。也就是说整个开发团队都可以自如地运行构建，而不需要一个专门的“构建负责人”做这件事。如果一个新加入的团队成员需要在周日的凌晨三点运行构建，他可以毫不犹豫地这么干
尽早反馈	一旦出了问题，你马上就能知道。在开发者处理需要集成的代码时这跟CI尤其相关
一致性	你知道部署的软件是什么版本，并且完全清楚每个版本的代码
依赖管理	大多数Java项目都有几个依赖项，比如log4j、Hibernate、Guice等。手工管理这些依赖项可能会非常困难，而且有一个版本发生变化就可能会导致软件不可用。良好的构建和CI能确保你总是针对同一个第三方依赖项进行编译和运行

为了将源码部署到运行时环境中，需要经过构建周期将其转变成二进制工件（JAR、WAR、RAR、EAR等）。比较老的Java项目通常都使用Ant，而比较新的则使用Maven或Gradle。很多开发团队还有夜间集成构建，有些已经升级成用CI服务器定期执行构建了。[③]

① 如果你的团队在谈论这些内容时或虔诚、或害怕，或话不多，那这一章就是为你准备的。
② 构建时间可配置：间隔可以是几分钟，也可以在提交时触发，或在其他特定时间运行。
③ 合作极其默契的项目团队能让非技术队友运行构建。

> **警告**　如果你从IDE中构建JAR文件或其他工件，那是在自找麻烦。从IDE中构建得到的不是与本地IDE设置无关的可重用构建，那简直就是埋下了祸根。作为朋友，我再怎么强调这一点都不为过：不允许你用IDE构建工件！

但大多数开发人员都觉得构建和CI不值得他们投入精力，他们觉得这个工作做起来不够爽，也得不到什么回报。构建工具和CI服务器经常是在项目一开始的时候搭起来，但很快就被遗忘了。这么多年来，我们听到过很多类似的说法："我们为什么还要在构建和CI服务器上花时间呢？现在弄得也挺好用的。够用就好，对不对？"

我们坚信良好的构建和CI能加快编码速度，提高代码质量。跟TDD（见第11章）相结合的构建和CI意味着你可以*毫无后顾之忧地快速重构*。你可以把它当做在你身后默默提供支持的导师，它为你营造一个安全的环境，让你可以快速编写并大胆修改代码。

本章，我们会首先介绍Maven 3。Maven 3是一个流行（还有争议，有些开发人员挺讨厌它）的构建工具，会强迫你按照严格定义好的构建周期工作。介绍Maven 3的内容中，除了常见的Java代码，还会涉及Groovy和Scala代码的构建。

Jenkins是CI界的流行天王，可以通过多种方式配置（以插件系统的方式）持续执行构建，并产生质量指标。在学习Jenkins时，我们还会深入了解FindBugs和Checkstyle产生的代码质量指标。

在学完Maven和Jenkins之后，你应该会彻底熟悉典型的Java构建和CI流程。之后我们会重点讨论Clojure的构建工具Leiningen，完全从另一个角度看看构建和部署工具。你会看到它如何在提供工业级强度的构建和部署能力的同时实现极其迅速、易用的TDD风格。

与Maven 3的相遇将开启你的构建和CI之旅！

12.1　与 Maven 3 相遇

Maven是流行的Java及JVM语言相关的构建工具，然而反对它的人和支持它的人态度同样坚决。它的设计理念是，严格的构建周期辅以强大的依赖管理是成功构建的必要条件。Maven不仅是构建工具，更是项目技术组件的管理工具。实际上，Maven的构建脚本叫做POM（Project Object Model，项目对象模型）文件。这些POM文件是用XML写的，并且每个Maven项目或模块都有一个pom.xml文件。

> **注意**　POM文件中马上要加入对备选语言的支持，从而满足用户对灵活性的要求（就像Gradle提供的那些功能）。

Ant和Gradle怎么样？

Ant是个流行的构建工具，特别是在早年的Java项目里。它作为公认的标准存在了相当长的一段时间。我们不准备在这里再讲了，因为之前已经有人讲过上百次了。更关键的是，我们

觉得Ant没有强制实行通用的构建周期，也没有一组通用（强制的）构建目标。这就是说开发人员必须研究手头每个Ant构建的细节。如果你要用Ant，Ant网站（http://ant.apache.org）列出了所有必需的细节。

Gradle是这一领域的新秀。它有意选择了和Maven相反的路线，限制不会那么严格，你可以按自己的方式声明构建过程。它也跟Maven一样提供依赖管理和很多其他特性。如果你想尝试下Gradle，可以访问Gradle网站（www.gradle.org）了解更多细节。

要学习优秀的构建实践，Maven是适合的工具。它强制你遵循Maven构建周期，一旦掌握这个构建周期，你就可以轻松地构建世界上任何一个Maven项目。

Maven采取了惯例优先配置的策略，并希望你能融入到它的世界观，在源码该怎么布局、属性如何过滤等设置上都能接受它的安排。这可能会吓着某些开发人员，但Maven的构建周期是经过多年深思熟虑总结出来的，沿着它提供的路径走往往是最合理的。而对于那些极力反对墨守成规的人，Maven确实提供了覆盖默认值的办法，但那样会做出更加繁琐，并且标准化程度更低的构建脚本。

用Maven执行构建就是让它执行一个或几个目标（代表特定任务，比如编译源码、运行测试等）。目标都是绑到默认构建周期中的，如果你要求Maven执行测试（如`mvn test`），它会先编译源码和测试代码。简言之，它会强迫你遵守正确的构建周期。

如果你还没装Maven 3，请参见附录A中的A.2节。在完成下载和安装之后，再回到这里来创建你的第一个Maven项目。

12.2 Maven 3 入门项目

Maven遵循惯例优先的原则，你只要创建一个快速启动项目，马上就能看到它惯用的项目结构。它喜欢的典型项目结构看起来和下面的布局类似。

```
project
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- company
    |               `-- project
    |                   `-- App.java
    |   `-- resources
    `-- test
        `-- java
            `-- com
                `-- company
                    `-- project
                        `-- AppTest.java
        `-- resources
`-- target
```

依照惯例，Maven把代码分成了main和test两部分。它还创建了一个特别的resources目录，构建工作所需的其他任何文件（比如用于日志的log4.xml文件、Hibernate配置文件以及其他类似资源文件）都放在这个目录下。pom.xml是Maven的构建脚本，关于这个文件的详情，请参见附录E。

如果你是多语言程序员，Scala和Groovy源码跟Java源码的结构一样，只是Java源码放在java目录下，而它们的根目录分别是scala和groovy。Java、Scala和Groovy代码可以高高兴兴地手拉手出现在同一个Maven项目中。

target目录是构建运行后才会创建的。所有的类、工件、报告和构建产生的其他文件都会出现在这个目录下。对于Maven项目结构的完整列表，请参见Maven网站上的Introduction to the Standard Directory Layout（标准目录布局介绍）页面（http://t.cn/aKJYxo）。

要为新项目创建这个结构，请执行下面的目标（注意其中的参数）：

```
mvn archetype:generate
    -DgroupId=com.mycompany.app
    -DartifactId=my-app
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DinteractiveMode=false
```

然后你会看到Maven开始刷屏，它在疯狂下载插件和第三方类库。Maven需要它们来运行这个目标，它的默认下载地址是Maven Central（工件的在线资源库）。

为什么Maven看起来像要把整个互联网都下载下来？

“哦，又来了，Maven又开始下载了。”这是构建Java项目的兄弟之间常说的模因[①]。但这真是Maven的错吗？我们认为它这样做有两个根本原因。一是第三方类库开发人员对包和依赖的管理很烂（比如在他们的pom.xml文件里指定一个实际上并不需要的依赖项）。另一个是继承自JAR为主的包系统自身的缺陷，没办法做更细化的依赖项控制。

除了“正在下载……”，控制台应该还会有下面这种声明：

```
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time: 1.703s
[INFO] Finished at: Fri Jun 24 13:51:58 BST 2011
[INFO] Final Memory: 6M/16M
[INFO] ------------------------------------------------------------------
```

如果这一步失败了，很可能是你的代理服务器不允许访问Maven Central，插件和第三方类库都放在那上面。要解决这个问题，只要编辑settings.xml文件（见附录A的A.2节），把下面这部分内容加上去，请根据你的实际情况为各元素填上恰当的值：

① 模因（Meme）也称为米姆、弥、弥因、弥母、迷因以及谜米等，是文化资讯传承单位。这个词是1976年理查德·道金斯在《自私的基因》一书中创造的，以生物学中的演化规则类比文化传承的过程。模因包含甚广，包括宗教、谣言、新闻、知识、观念、习惯、习俗，甚至口号、谚语、用语、用字、笑话。——译者注

```
<proxies>
  <proxy>
    <active>true</active>
    <protocol></protocol>
    <username></username>
    <password></password>
    <host></host>
    <port></port>
  </proxy>
</proxies>
```

重新运行上面的目标，这次应该能看到my-app项目出现在了目录中。

> **提示** 如果团队中的所有人都遇到了这个问题，请在$M2_HOME/conf/settings.xml中加上代理配置。

Maven支持的原型（项目布局）几乎是无限的。如果要生成某个特定类型的项目（比如JEE6的项目），可以执行`mvn archetype:generate`目标，然后只要遵照它给你的提示就行了。

为了探索Maven的更多细节，我们来看一个源码和测试代码都已经准备好的项目，用它把整个构建周期走一遍。

12.3 用 Maven 3 构建 Java7developer 项目

还记得图12-1中的构建周期吗？Maven的构建周期跟那个类似，你马上就要经历构建周期中的每个阶段了。尽管本书中的源码不是一个应用程序，我们还是会把它们统一放到一个叫做java7developer项目中。

这一节的重点是：

- 探索Maven POM文件（即构建脚本）的基础；
- 如何编译、测试和打包代码（包括Scala和Groovy）；
- 如何用环境配置处理多个环境；
- 如何生成一个包含各种报告的项目网站。

首先你要搞明白定义java7developer项目的pom.xml文件。

12.3.1 POM

java7developer项目用pom.xml表示，包括各种插件、资源，以及构建所需的其他元素。可以在解压或签出本书项目代码的根目录（从现在开始我们用$BOOK_CODE指代这个位置）中找到这个pom.xml文件。POM主要由四部分组成：

- 项目基本信息；
- 构建配置；
- 依赖项管理；
- 环境配置。

这是个相当长的文件，但实际上它没有看起来那么复杂。如果你想了解POM中可以包含哪些内容的完整细节，请参见Maven网站上的POM Reference（http://maven.apache.org/pom.html）。

接下来我们就要解释java7developer项目pom.xml文件的这四部分，先从项目基本信息开始。

1. 项目基本信息

pom.xml文件中可以放入一系列的基本项目信息。代码清单12-1列出的是最起码的起步信息。

代码清单12-1　项目基本信息

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.java7developer</groupId>
  <artifactId>java7developer</artifactId>          ❶ 唯一标识
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>java7developer</name>
  <description>                                    ❷ 项目信息
    Project source code for the book!
  </description>
  <url>http://www.java7developer.com</url>

  <properties>
    <project.build.sourceEncoding>
      UTF-8                                        ❸ 平台无关的字符编码
    </project.build.sourceEncoding>
  </properties>
  ...
```

这个工件在Maven资源库中的唯一标识符由三部分组成：第一部分是`<groupId>`的值`com.java7developer`❶；第二部分是`<artifactId>`的值`java7developer`。`<packaging>`的值`jar`告诉Maven你要构建一个JAR文件（这里可能出现的值有`war`、`ear`、`rar`、`sar`和`har`）。唯一标识的最后一部分是`<version>`的值`1.0.0`[①]，表明版本号（执行Maven发布时会在这个值后面加上`SNAPSHOT`）。

文件中还指定了`<projectName>`和`<url>`，以及一些其他可选的项目信息❷。`<sourceEncoding>`为UTF-8，这样可以确保在所有平台上的构建都是一致的❸。

总的来说，这个配置会指导Maven构建出java7developer-1.0.0.jar工件，并把它存在Maven资源库中的com/java7developer/1.0.0目录下。

Maven版本和快照

作为Maven惯例优先原则的一部分，它倾向于以主要.次要.琐碎的格式来设置版本号，并依照惯例在版本号后面加上`-SNAPSHOT`表示这是一个临时性的工件。比如说，在你的团队为

① 版本号的格式遵循Major.Minor.Trivial风格，这是我们的最爱！

即将发布的1.0.0版本持续构建JAR时，Maven会依照惯例将版本号设置为1.0.0-SNAPSHOT。这样，各种Maven插件就知道这还不是生产版本，从而可以正确处理它。在把这个工件发布到生产环境中时，要发布为1.0.0，下一个修订bug的版本要从1.0.1-SNAPSHOT开始。

Maven通过它的发布插件把这些都自动化了。要了解更多细节，请参见发布插件页面（http://maven.apache.org/plugins/maven-release-plugin/）。现在你已经明白项目基本信息部分是什么样的了，接下来我们来看看<build>吧。

2. 构建配置

<build>中包含执行Maven构建周期目标所需的插件[①]及相应的配置。在大多数项目中，这部分内容都相当少，因为通常用默认插件的默认设置就够了。

在java7developer项目中，<build>中有几个覆盖了默认值的插件，以便可以：

- 构建Java 7代码；
- 构建Scala和Groovy代码；
- 运行Java、Scala和Groovy测试；
- 提供Checkstyle和FindBugs代码指标报告。

插件是以JAR为主的工件（主要是用Java写的）。要配置构建插件，需要把它放在pom.xml文件的<build><plugins>中。跟所有Maven工件一样，每个插件都有唯一标识，所以需要指定<groupId>、<artifactId>和<version>信息。对插件的所有配置都放在<configuration>中，并且每个插件的具体配置元素是不同的。比如编译插件的配置元素有<source>、<target>和<showWarnings>，这是编译器独有的配置信息。

代码清单12-2列出的是java7developer项目的构建配置部分（完整的代码清单及相应的解释在附录E中）。

代码清单12-2　POM：构建信息

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <showDeprecation>true</showDeprecation>
        <showWarnings>true</showWarnings>
        <fork>true</fork>
```

❶ 所用插件

❷ 编译Java 7代码

❸ 编译器警告

① 如果你需要对构建进行配置，可以访问Maven的插件页面查看插件的完整列表（http://maven.apache.org/plugins/index.html）。

```
          <executable>${jdk.javac.fullpath}</executable>      ◁── ❹ javac的路径
        </configuration>
      </plugin>

      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.9</version>
        <configuration>
          <excludes>
            <exclude>
              com/java7developer/chapter11/
➥listing_11_2/TicketRevenueTest.java
            </exclude>
            <exclude>                                          ❺ 排除的测试
              com/java7developer/chapter11/
➥listing_11_7/TicketTest.java
            </exclude>
            ...
          </excludes>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

因为Maven 3默认是编译Java 1.5的代码，而我们要编译Java 1.7❷，所以需要指明编译器插件（的版本）❶。

既然你打破了惯例，所以还要加上几个编译警告选项❸。接下来要指定Java 7安装在哪儿❹。只需要把sample_<os>_build.properties文件另存为build.properties，并编辑其中的`jdk.javac.fullpath`属性，因为Maven会用到它。

Surefire插件是测试用的。在配置中我们排除了几个失败测试❺（有两个第11章的TDD测试）。

现在构建部分已经讲完了，可以进入POM中非常重要的部分了：依赖管理。

3. 依赖管理

大多数Java项目的依赖项列表都很长，java7developer项目也不例外。Maven Central Repository中有各种各样的第三方类库，所以Maven可以帮你管理这些依赖项。最重要的是，这些第三方类库都有它们自己的pom.xml文件，会声明各自的依赖项，Maven可以据此找出任何需要下载的其他类库。

这些依赖项一开始主要分为两个作用域（`compile`和`test`）[①]。设置作用域跟把JAR文件放到CLASSPATH下是一样的效果。代码清单12-3是java7developer项目的`<dependencies>`部分。完整的代码清单及相应的解释在附录E中。

代码清单12-3　POM：依赖项

```
<dependencies>

  <dependency>
```

① J2EE/JEE项目通常也会用到runtime作用域的依赖项。

```
      <groupId>com.google.inject</groupId>
      <artifactId>guice</artifactId>
      <version>3.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>javax.inject</groupId>
      <artifactId>javax.inject</artifactId>
      <version>1</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <version>1.8.5</version>
      <scope>test</scope>
    </dependency>
    ...
  </dependencies>
```

❶ 工件的唯一ID

❷ 编译作用域

❸ 测试作用域

为了让Maven找到你引用的工件，需要给它正确的`<groupId>`、`<artifactId>`和`<version>`❶。我们在之前提到过，把`<scope>`设置为`compile`❷会把这些JAR加到CLASSPATH中用于代码的编译。将`<scope>`设置为`test`❸会在Maven编译和运行测试时把这些JAR加到CLASSPATH中。

但你怎么知道该指定什么`<groupId>`、`<artifactId>`和`<version>`？答案是搜索Maven Central Repository（http://search.maven.org/），你几乎总能找到答案。

如果找不到合适的工件，可以用`install:install-file`目标自己手工下载和安装插件。这里有个安装asm-4.0_RC1.jar类库的例子。

```
mvn install:install-file
-Dfile=asm-4.0_RC1.jar
-DgroupId=org.ow2.asm
-DartifactId=asm
-Dversion=4.0_RC1
-Dpackaging=jar
```

这个命令运行完后，你应该能在本地资源库的$HOME/.m2/repository/org/ow2/asm/asm/4.0_RC1/中找到安装好的工件，就像Maven下载的一样。

工件管理器

在你手工安装一个第三方类库时，你只是为自己装的，团队里的其他人呢？在你做要跟同事共享的工件时也面临相同的问题，但你也不能把它放到Maven Central中（因为那是你们的私有代码）。

用二进制工件管理器可以解决这个问题，比如Nexus（http://nexus.sonatype.org/）。工件管理器就像你和团队自有的本地Maven Central，外界无法访问。大多数工件管理器还会缓存Maven Central和其他资源库，你的开发团队所需的依赖项都可以从它那里得到。

环境配置是要搞懂的最后一部分POM了，它可以有效处理不同环境下的构建。

4. 环境配置

环境配置是Maven用来处理环境化（比如UAT跟生产环境之间的构建差异）或其他与普通构建稍有不同的构建变体的。java7developer项目中有个例子，其中一个环境配置会关闭编译器和作废警告，如代码清单12-4所示。

代码清单12-4　POM：环境配置

```
<profiles>
  <profile>
    <id>ignore-compiler-warnings</id>          <-- ❶ 该环境配置的ID
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.3.2</version>
          <configuration>
            <source>1.7</source>
            <target>1.7</target>
            <showDeprecation>false</showDeprecation>     | ❷ 关闭警告
            <showWarnings>false</showWarnings>           |
            <fork>true</fork>
            <executable>${jdk.javac.fullpath}</executable>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

在执行Maven时用`-P <id>`可以指定要启用的环境配置（比如`mvn compile -P ignore-compile-warnings`）❶。在这个环境配置被激活后，会使用指定的编译器插件，作废警告和其他编译器警告都会被关闭❷。

在Introduction to Build Profiles（构建环境配置介绍）页面可以找到更多关于环境配置和为其他环境化目的如何使用它们的信息（http://maven.apache.org/guides/introduction/introduction-to-profiles.html[①]）。

终于完成了java7developer项目的pom.xml文件之旅，你是不是已经迫不及待地想构建它了？

① 短链接：http://t.cn/zl7MyO1。——译者注

12.3.2 运行示例

希望你已经把代码下载下来了。你会在其中看到一些pom.xml文件，就是它们控制着Maven构建。

你会在这一节中经历最常用的Maven构建周期目标（clean、compile、test和install）。第一个目标就是清除上次构建遗留的所有工件。

1. 清除

目标clean会删掉target目录。请换到$BOOK_CODE目录并执行clean目标。

```
cd $BOOK_CODE
mvn clean
```

与你要用到的其他Maven构建目标不同，clean不会自动调用。如果想清除上次构建的工件，必需手动加上clean目标。

上次构建遗留的残余物已经清除了，接下来一般是执行编译源码的构建目标。

2. 编译

目标compile用pom.xml文件中的编译器插件配置编译在src/main/java、src/main/scala和src/main/groovy下的源码。也就是说它会带着加到CLASSPATH中的编译作用域依赖项执行Java、Scala和Groovy编译器（javac、scalac和groovyc）。Maven还会处理在src/main/resources目录下的资源文件，确保它们作为编译CLASSPATH的一部分。

编译后的类最终会出现在target/classes目录下。请执行下面的Maven目标：

```
mvn compile
```

compile目标的执行应该相当快，并且在控制器中应该有类似下面这种输出。

```
...
[INFO] [properties:read-project-properties {execution: default}]
[INFO] [groovy:generateStubs {execution: default}]
[INFO] Generated 22 Java stubs
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 119 source files to
     C:\Projects\workspace3.6\code\trunk\target\classes
[INFO] [scala:compile {execution: default}]
[INFO] Checking for multiple versions of scala
[INFO] includes = [**/*.scala,**/*.java,]
[INFO] excludes = []
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\java:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\target\generated-sources\groovy-
     stubs\main:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\groovy:-1: info:
     compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\scala:-1: info: compiling
[INFO] Compiling 143 source files to
     C:\Projects\workspace3.6\code\trunk\target\classes at 1312716331031
[INFO] prepare-compile in 0 s
```

```
[INFO] compile in 12 s
[INFO] [groovy:compile {execution: default}]
[INFO] Compiled 26 Groovy classes
[INFO]-----------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ----------------------------------------------------------------
[INFO] Total time: 43 seconds
[INFO] Finished at: Sun Aug 07 12:25:44 BST 2011
[INFO] Final Memory: 33M/79M
[INFO] ----------------------------------------------------------------
```

在这一阶段，src/test/java、src/test/scala和src/test/groovy目录下的测试类还没编译。尽管专门有一个test-compile目标编译这些类，但最常见的方式是运行test目标。

3. 测试

在目标test中能见到Maven构建周期的实际效果。在你要求Maven运行测试目标时，它知道为了保证test目标成功运行，需要把前面的所有构建周期目标都执行一下（包括compile、test-compile和一系列其他目标）。

Maven会通过神火（Surefire）插件，使用pom.xml文件中的测试提供者（作为测试作用域的依赖项，此例中为JUnit）运行测试。Maven不仅会运行测试，还会产生报告文件，测试完成后你可以分析这些报告，以便对失败测试展开调研，并收集测试指标。

请执行下面的Maven命令：

```
mvn clean test
```

一旦完成测试类的编译和运行，应该就能见到下面这种输出。

```
...
Running com.java7developer.chapter11.listing_11_3.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_4.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_5.TicketTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec

Results :

Tests run: 20, Failures: 0, Errors: 0, Skipped: 0

[INFO]-----------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]-----------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Wed Jul 06 13:50:07 BST 2011
[INFO] Final Memory: 24M/58M
[INFO]-----------------------------------------------------------------
```

测试结果存在target/surefire-reports目录下。现在你可以去看看那里的文本文件。稍后你能在一个更棒的Web页面上看到这些结果。

提示　你应该注意到了，命令中还有clean目标。我们这么做是出于习惯，以防有遗留的残余物欺骗我们。

现在你的代码编译过，也测试过了，并且已经准备好打包了。尽管你可以直接用package目标，但我们会用install目标。想知道为什么，且看下文分解！

4. 安装

目标install的任务主要有两个。按pom.xml文件中<packaging>指定的方式（此例中为JAR文件）对编译结果打包。然后把打包好的工件安装到本地Maven资源库中（在$HOME/.m2/repository下），以便其他项目可以把它当做依赖项用。跟其他目标一样，如果它发现之前的构建步骤还没执行，它也会先执行这些相关目标。请执行下面的Maven命令：

```
mvn install
```

一旦完成install目标，你应该见到下面这种输出报告。

```
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\Projects\workspace3.6\code\trunk\target\
java7developer-1.0.0.jar
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\Projects\workspace3.6\code\trunk\target\java7developer-
     1.0.0.jar
to C:\Documents and Settings\Admin\.m2\repository\com\java7developer\
java7develope
r\1.0.0\java7developer-1.0.0.jar
[INFO]------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]------------------------------------------------------------------
[INFO] Total time: 17 seconds
[INFO] Finished at: Wed Jul 06 13:53:04 BST 2011
[INFO] Final Memory: 28M/66M
[INFO]------------------------------------------------------------------
```

在target目录（package目标的结果）和本地Maven资源库的$HOME/.m2/repository/com.java7developer/1.0.0目录下应该能找到java7developer-1.0.0.jar工件。

提示 你可能希望把Scala和Groovy代码分别放到它们自己的JAR文件中。Maven支持这种操作，但你必须记住，对于Maven来说，每个独立的JAR工件都应该对应一个独立的项目。也就是说你必须用到Maven中多模块项目的概念。请参见Maven的Guide to Working with Multiple Modules（多模块处理指南）页面了解细节（http://maven.apache.org/guides/mini/guide-multiple-modules.html[①]）。

我们大多数人都是在团队中工作，并且经常要共享代码库，那我们怎么才能保证快速、可靠地构建大家共享的代码呢？这要靠CI服务器来保证，并且到目前为止，对于Java开发人员来说现在最流行的CI服务器是Jenkins。

① 短链接http://t.cn/zjIlX70。——译者注

12.4　Jenkins：满足 CI 需求

CI的成功要靠管理（开发纪律）和工具相结合。为了让CI过程达到优秀的标准，Jenkins提供了很多必需的支持，如表12-2所示。

表12-2　衡量CI构建是否优秀的标准及Jenkins如何达成这些标准

标　准	Jenkins如何达成
自动构建	Jenkins会在你需要的任何时间运行构建。它可以通过构建触发器实现自动构建
一直测试	Jenkins能运行任何你想要的目标，包括Maven的`test`。它有强大的测试失败趋势报告，只要有一个测试没通过，它就会报告构建失败
定期提交	这是开发人员的事
每次提交都构建	每次检测到版本控制库的新提交时Jenkins都可以执行构建
快速构建	这对基于单元测试的构建更加重要，因为你想要它们有更快的往返时间。Jenkins可以把工作发送给从属节点从而提高速度，但更主要的是开发人员做出精益的、有意义的构建脚本，并配置Jenkins在执行构建时调用恰当的构建周期目标
结果可视化	Jenkins有基于Web的仪表板，还有一套发送通知的办法

所有CI服务器都能轮询版本控制资源库，并执行构建周期目标`compile`和`test`。让Jenkins脱颖而出的是它易于使用的UI和可扩展的插件生态系统。

在配置Jenkins和它的插件时，UI的帮助非常大，它经常会在你输入完成后用Ajax检查输入的有效性。它还提供了大量的情景式帮助信息，运行Jenkins根本就不需要专业技能。

Jenkins的插件包罗万象，几乎可以轮询任何版本控制资源库，并且可以生成一系列非常有价值的代码报告。

Jenkins和Hudson

在网上和某些书中，这个CI服务器的名字有些混乱。Jenkins实际上是Hudson项目最近出现的一个副本，主流的开发人员和活跃的社区现在都集中在Jenkins上。Hudson本身仍然是一个优秀的CI服务器，但相较而言Jenkins项目更活跃。

Jenkins是自由的开源软件，其社区充满活力，对新手帮助很大。

关于如何下载和安装Jenkins，请参阅附录D。完成下载和安装后，马上回来继续！

警告　假定你会把Jenkins的WAR文件装到Web服务器上，那么Jenkins安装的根URL是http://localhost:8080/jenkins/。如果是直接运行WAR文件[①]，根URL应该是http://localhost:8080/。

本节会讨论Jenkins安装的基础配置，然后是如何设置、执行构建任务。我们会以java7developer

① 指运行java –jar Jenkins.war。——译者注

项目为例，但你可以随意选用自己喜欢的项目。

为了让Jenkins监测源码资源库并执行构建，需要先配置基础设置。

12.4.1 基础配置

我们会从Jenkins的主页http://localhost:8080/jenkins/开始。要配置Jenkins，请点击左边菜单中的Manage Jenkins（管理Jenkins）链接（http://localhost:8080/jenkins/manage）。管理页中列出了很多设置选项。

现在，选择Configure System（系统配置）链接（http://localhost:8080/jenkins/configure）。你应该能进入类似于图12-2的界面中。

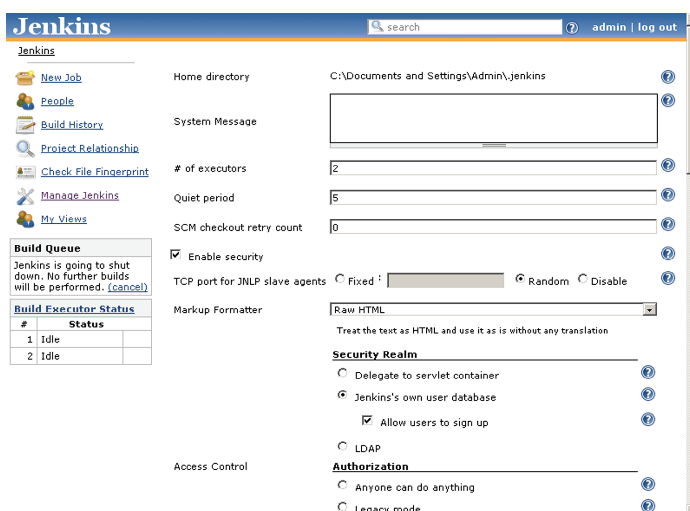


图12-2 Jenkins配置页

从界面的顶部可以看到Jenkins的home目录的位置。如果需要在UI之外进行配置，可以到这个目录中去。

提示 如果你是为团队安装Jenkins，并且需要考虑安全性，应该选中Enable Security（安全保护）和Prevent Cross Site Request Forgery Exploits（阻止跨域攻击请求）多选框，并进行相应的配置。对初学者来说，用Jenkins自身的数据库最容易。以后你可以随时切换到企业LDAP上，或基于Active Directory（活动目录）进行认证和授权。

为了执行构建，Jenkins需要知道构建工具放在哪里。这还要在配置页中设置，找到单词“Maven”。

1. 构建工具配置

Jenkins内置了对Ant和Maven（可以用插件支持其他构建工具）的支持。在java7developer项

目中，我们用的是Maven（在Windows上），所以对Jenkins的配置如图12-3所示。

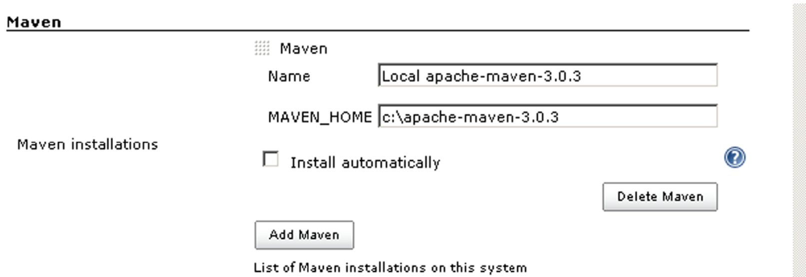


图12-3　设置构建工具Maven

注意，Jenkins有一个自动安装Maven的选项，在没装Maven的机器上，这个选项还是挺方便的。

现在Maven配置好了，需要告诉Jenkins你用什么版本控制资源库。这在配置页的下面。找到单词SVN。

2. 版本控制配置

Jenkins内置了对CVS和Subversion（SVN）的支持。像Git和Mercurial这样的版本控制系统也有插件。java7developer项目用SVN 1.6，配置如图12-4所示。

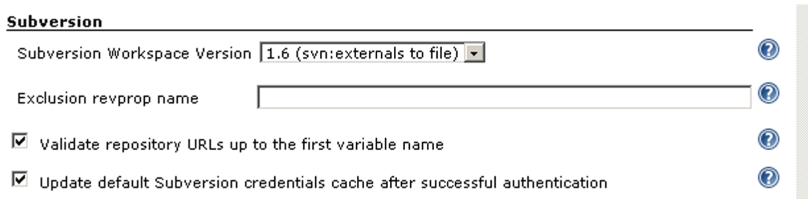


图12-4　SVN版本控制配置

在设置好这些配置后，点击屏幕底部的Save按钮，以确保这些配置会被保存下来。

现在Jenkins的基础设置已经做好了，可以创建你的第一个任务了。

12.4.2　设置任务

要设置新任务，请回到仪表板中并点击左手菜单的New Job（新任务）链接，进入任务设置页面（http://localhost:8080/jenkins/view/All/newJob）。这里有很多选项可供选择。

要设置一个任务来构建java7developer项目，先要给任务确定一个标题（java7developer），选择Build a Maven 2/3 Project（构建一个Maven 2/3项目）选项并点击OK按钮继续。你应该会进入一个类似图12-5的配置界面。这里有一些输入项要填，但下面这些应该是你先填好的内容：

- 源码管理；
- 构建触发器；
- 构建。

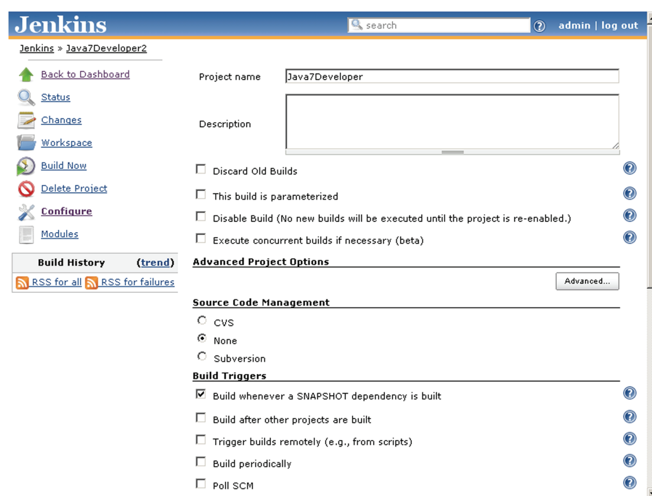


图12-5　Maven 2/3任务配置页面

我们从源码管理的配置开始。

1. 源码管理

源码管理主要设置要构建的源码来自版本控制的哪个分支、标记或标签。随着你的团队向版本控制系统中稳步添加源码，它就是持续集成中的“集成”。对于java7developer项目，我们用SVN构建主干中的源码。设置如图12-6所示。

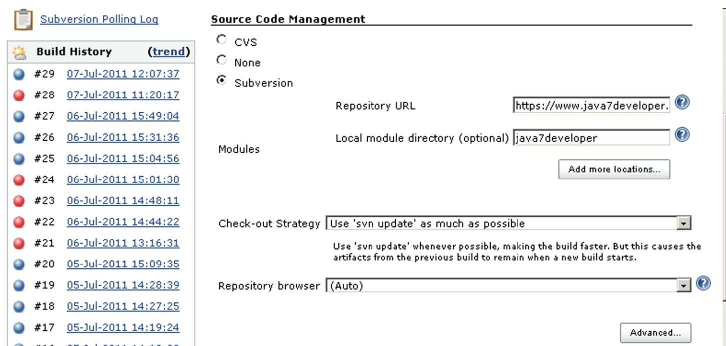


图12-6　java7developer源码管理配置

一旦告诉Jenkins从哪里获取源码，接下来要配置的就是Jenkins应该隔多长时间构建一次，这是通过构建触发器完成的。

2. 构建触发器

构建触发器把“持续”引入了持续集成。你可以要求Jenkins在源码控制库每次有新提交时就进行构建，或者采用更悠闲的方式，设为每日构建一次。

我们对java7developer项目的设置，只是要求Jenkins每隔15分钟轮询SVN一次，如图12-7所示。

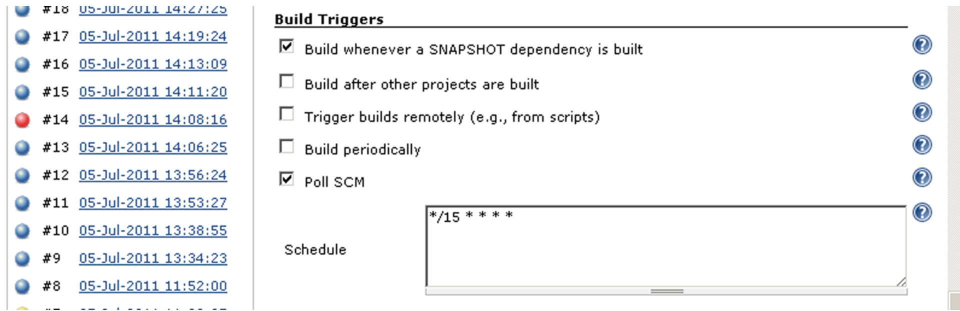


图12-7　java7developer构建触发器配置

你可以点击输入项旁边的帮助图标（表示为？）查看帮助信息。在这个例子中，编写类似cron的表达式来指定轮询周期时你可能需要帮助。

到这个阶段，Jenkins已经知道了到哪里去找源码，隔多长时间构建一次。接下来就该告诉Jenkins应该执行哪个构建阶段（构建脚本中的目标或目的）。

3. 构建

用Jenkins可以设置很多任务来执行构建周期的不同阶段。你可能想要一个每晚执行一次完整的系统集成测试的任务。但更多情况下，你可能想要执行频率更高的任务，在每次有新的源码提交到版本控制系统时编译源码并运行单元测试。

对于java7developer项目，我们要求Jenkins执行Maven的`clean`和`install`目标，如图12-8所示。

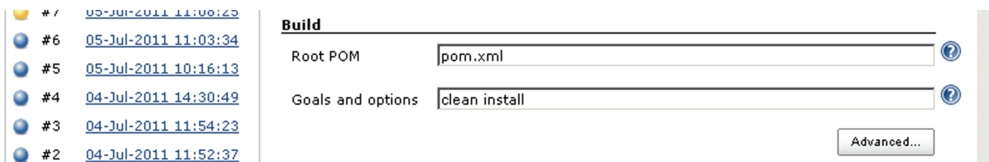


图12-8　Java7developer任务中要执行的Maven构建目标（`clean`、`install`）

Jenkins现在有java7developer项目的所有信息了，它可以每隔15分钟轮询一次SVN代码库的主干，执行Maven的`clean`和`install`目标。不要忘了点击Save按钮把任务存下来！

现在可以回到仪表板，在那里看看你的任务，它应该非常像图12-9。

在Last Success (S)一栏（最近一次成功），圆形图标表示任务最后一次构建的状态。在Weather (W)（天气）一栏，天气图标表示项目的总体健康状况，它是由构建失败的频率、测试是否通过，还有一系列其他可能情况（取决于你配置的插件）决定的。要进一步了解这些图标的含义，请点击仪表板中的Legend（图例）链接（http://localhost:8080/jenkins/legend）。

现在任务已经准备好了，你可能想看看它运行起来是什么样！你可以等15分钟后的第一次轮询，也可以强制执行一次构建。

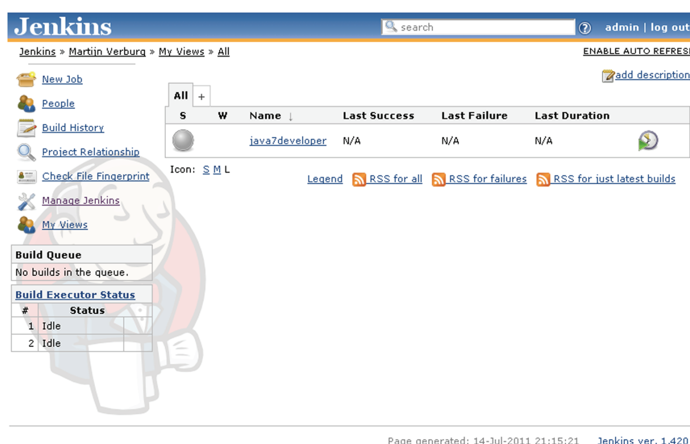


图12-9　有java7developer任务的仪表板

12.4.3　执行任务

要检查新配置，强迫任务执行是个好办法。对于java7developer任务，只要到仪表板中，点击Schedule a Build（调度构建）按钮（在Last Duration旁边带绿色箭头的钟表图标）。然后你就能刷新页面去看看构建的执行结果了。

提示　点击仪表板右上角的Enable Auto Refresh（允许自动刷新），可以让仪表板自动刷新。这样你就可以及时看到当前构建的状态了。

在构建执行时，java7developer任务的第一个图标会闪，表明构建正在处理中。在页面左侧还能看到Build Executor Status（构建执行器状态）。构建一完成，Last Success (S)（最近一次成功）一栏那个圆形图标就变成了红色，表明构建失败了。

这个失败是因为漏掉了build.properties文件。如果你阅读12.2节时没按照书中的指示做，现在也可以很快解决掉这个问题，找到build.properties文件样本，编辑它，以便能找到要用到的Java 7 JDK。下面是在Unix上执行这些操作的例子：

```
cd $USER/.jenkins/jobs/java7developer/workspace/java7developer
cp sample_build_unix.properties build.properties
```

现在你可以回到仪表板中，再次手工运行构建。这次构建应该能成功，并且仪表板中java7developer任务的Last Success（最近一次成功）一栏应该是个蓝色图标，表示构建成功了。

你还可以马上去看一下构建的测试报告，因为Jenkins知道如何读取Maven产生的输出。要看测试结果，可以点击java7developer任务Last Success一栏的链接（http://localhost:8080/jenkins/job/java7developer/lastSuccessfulBuild/）。该链接会将你带入Latest Test Result（最新测试结果）页面，其界面如图12-10所示。

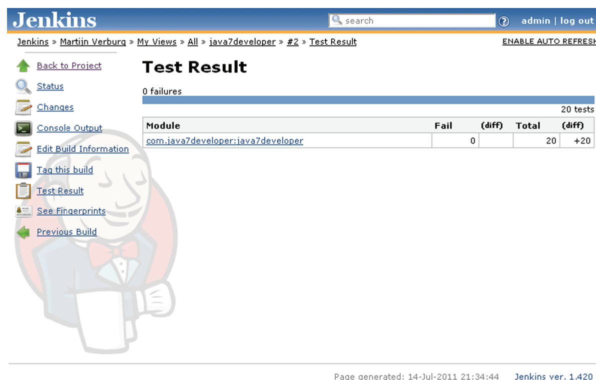


图12-10　java7developer成功构建的测试结果

测试全部通过，非常棒！如果有任何一个失败，你可以深入了解每个测试的细节。

我们对运行失败和成功构建的基础都进行了总结。对于java7developer项目来说，Jenkins会继续轮询SVN并在发现新提交时执行新构建。

你已经见过Jenkins如何运行构建，构建因不同原因失败时如何在界面上发出警告，如何检查测试成功或失败。但Jenkins可以做得不止这些，它还能给出一系列实用的代码指标，让你对代码的质量有更深入的认识。

12.5　Maven 和 Jenkins 代码指标

Java和JVM已经存在很长时间了，并且这么些年积累下来，已经开发出了一些强大的工具和类库来指导开发人员编写优质的代码。我们宽泛地把这个领域定义为代码指标或静态代码分析。Maven和Jenkins都支持目前最流行的工具。尽管这些流行的工具（或新的专门化工具）对其他语言的支持越来越多，但它们仍然主要是针对Java语言自身。

提示　现代IDE（比如Eclipse、IntelliJ和NetBeans）也支持几种静态语言分析工具和类库，值得花些时间研究一下。

代码指标工具主要是为了消除所有开发人员都会犯的小错误。它能帮你树起最低的代码质量标杆，告诉你下面这些情况：

- 测试对代码的覆盖率[①]；
- 代码的格式是否清晰（有助于差异比较和可读性）；
- 是否很可能会出现NPE；
- 是否忘记了域对象中的`equals()`和`hashCode()`方法。

① 本书不讨论普通的代码覆盖工具，因为它们和Java 7还不兼容。

各种工具提供的检查列表都很长，但开发团队要自己决定对项目做哪些检查。

代码指标的局限性

一些团队因为解决了代码指标工具警告过的问题就认为他们的代码库臻于完善了，这是不对的。代码指标警告能帮你去掉很多低级bug，避免糟糕的编码实践。但它们不能保证代码质量，或者判断业务逻辑实现是否正确。

另外一个问题是管理层可能想把这些指标放到报告里。为了管理层和你们自己考虑，请把代码指标留在开发人员这一层面。它们不是项目管理指标。

Maven和Jenkins结合得很好，既可以提供代码指标的概览，也能告诉你其中的细节。本节的两个主要内容如下：

- 如何安装并配置Jenkins插件；
- 如何配置代码一致性（Checkstyle）和bug查找（FindBugs）插件。

我们仍以java7developer为例。先来看看如何安装Jenkins插件，这是得到代码指标报告功能的前提条件。

12.5.1　安装Jenkins插件

Jenkins基于UI的插件管理器很不错，它可以帮你下载和安装插件，所以安装Jenkins插件并不复杂。安装Jenkins插件时需要重启Jenkins，所以应该先进入Jenkins管理页（http://localhost:8080/jenkins/manage），并点击Prepare to Shutdown（准备关闭）链接。这会终止所有即将执行的任务，以便你可以安全地安装插件，重启Jenkins。

Jenkins准备好关闭之后，就可以访问插件管理器了。在管理页中点击Manage Plugins（管理插件）链接（http://localhost:8080/jenkins/pluginManager/）。应该能见到如图12-11所示的界面。

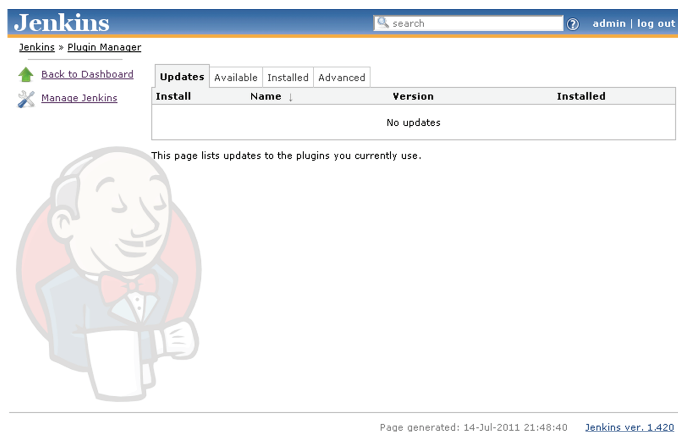


图12-11　Jenkins插件管理器

第一个是Updates（更新）标签。切换到Available（可用）标签，能见到一长串可用插件的列表。为学习本章内容，需要选中下面这些插件：

- Checkstyle；
- FindBugs。

然后点击界面底部的Install按钮开始安装。安装完成后，可用通过链接http://localhost:8080/jenkins/restart重启Jenkins。

重启Jenkins之后插件就安装好了。现在该去配置这些插件了，先从Checkstyle插件开始。

12.5.2　用Checkstyle保持代码一致性

Checkstyle是静态代码分析工具，主要关注代码布局、Javadoc层次是否恰当，以及其他语法糖的检查。它还会检查常见的代码错误，但FindBugs检查得更加全面。

Checkstyle的重要性体现在两个方面。首先，它有助于强化小组的编码风格规范，以便团队成员可以很容易地读懂彼此的代码（易读性是Java得以流行的一个主要原因）。其次，如果代码元素的位置和空格保持一致，diffs和patches用起来就更容易了。

我们在Maven的pom.xml文件中已经配置过Checkstyle插件了，所以你只需要在java7developer任务中加上`checkstyle:checkstyle`目标。要配置该任务，点击在仪表板中列出的java7developer链接，然后在新界面上点击左侧菜单中的Configure（配置）链接。

接着配置报告，并确定违规的情况出现次数太多时是否应该放弃构建。图12-12中是我们对java7developer项目中Maven构建及报告的配置。

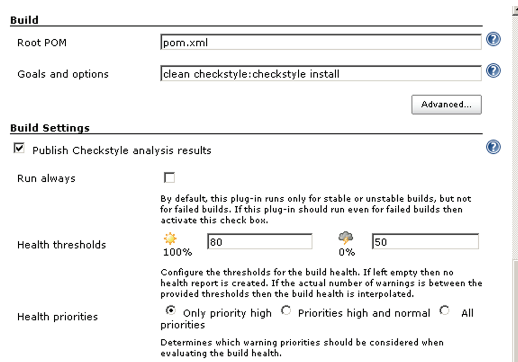


图12-12　Checkstyle配置

别忘了点击Save把这个配置存下来！Checkstyle的默认规则是Sun公司最初提出来的Java编码规范。Checkstyle能微调到第*n*级，所以应该能准确表示团队的编码规范。

警告　最新版的Checkstyle还没全面支持Java 7语法，所以你见到对try-with-resources、钻石语法和其他Coin项目语法的肯定可能是假的。

让我们来看看默认规则集如何把Java7developer项目组织起来。跟往常一样，你可以回到Jenkins的仪表板中，手工执行构建。构建完成后，回到最近构建成功页面（记住，可以通过Last Success一栏的链接访问该页面），点击左侧菜单上的Checkstyle Warnings（Checkstyle警告）链接进入Checkstyle报告页。java7developer项目的Checkstyle报告页看起来应该如图12-13所示。

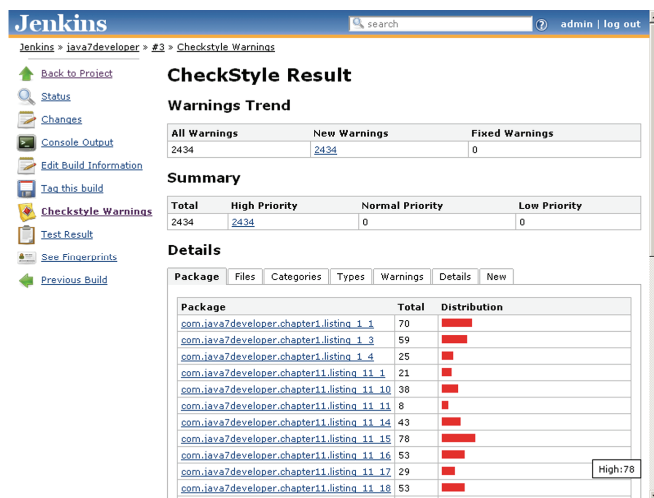


图12-13　Checkstyle报告

如你所见，在Java7developer的代码上有些警告信息。看起来我们还有些工作要做！你可以深入到每个警告中，看看为什么会发生违规，并在下一次构建之前改正它。

Checkstyle肯定在这方面有所帮助，但它的重点不是潜在的代码错误。这种重要的代码错误检查最好用FindBugs插件。

12.5.3　用FindBugs设定质量标杆

FindBugs（Bill Pugh出品）是为了找出代码中潜在的bug而做的字节码分析工具。由于它分析的是字节码，所以也能用在Scala和Groovy上。但因为它所设置的规则是为了捕获Java语言中的bug，所以如果你用它来分析Groovy和Scala代码，要对其持审慎的态度，因为它可能发现不了其中的bug。

FindBugs背后有大量研究成果的支持，都是由特别熟悉Java语言的开发人员做的。它能检测出下面这些状况：

- ❑ 会导致NPE的代码；
- ❑ 赋值给一个从来没用到的变量；
- ❑ 用==而不是用`equals`方法比较字符串对象；
- ❑ 在循环中用基本的+操作符而不是用`StringBuffer`合并字符串。

你应该先试试FindBugs的默认设置，然后再根据要检测的规则进行微调。

警告　即便是Java语言，FindBugs也会误报。你应该认真研究它发出的警告，如果确信可以忽略它们，可以显式排除这些情况。

FindBugs的重要性体现在两个方面。首先，它以结对程序员的角色教开发人员养成好习惯（尽可能帮助检测出潜在bug）。其次，项目的总体质量变好了，并且问题跟踪单里不会再充满烦人的小bug，让团队可以解决实际问题，比如业务逻辑的变化。

跟Checkstyle插件一样，你可以点击仪表盘任务列表中的java7developer链接，然后在新界面中点击左侧菜单上的Configure链接。

为了执行FindBugs插件，还要在Jenkins中添加Maven构建目标`compile findbugs:findbugs`（需要`compile`以便FindBugs能处理字节码）。

除了定义违例过多构建是否失败，还可以配置报告。如图12-14所示。

图12-14　FindBugs配置

别忘了点击Save把这个配置存下来！FindBugs预定义的规则集可以大范围调整，以准确表示团队的编码规范。让我们来看看默认规则集如何应用到Java7developer项目上。

跟往常一样，你可以回到Jenkins的仪表盘中，手工执行构建。构建完成后，回到最近构建成功页面（记住，可以通过Last Success栏的链接访问该页面），点击左侧菜单上的FindBugs Warnings链接进入FindBugs报告页。java7developer项目的FindBugs报告页看起来应该如图12-15所示。

如你所见，Java7developer的代码有些警告信息。写书的作者也可能写出不完美的代码！你可以仔细检查每个警告，看看为什么会发生违规，并且如果你愿意，可以在下一次构建之前改正它。

FindBugs会把大部分常见的Java陷阱和编码错误都找出来。随着开发团队对这些错误的了解程度不断加深，报告中的警告数量会逐步减少。你们不仅提升了代码的品质，还完善了自身的编码能力！

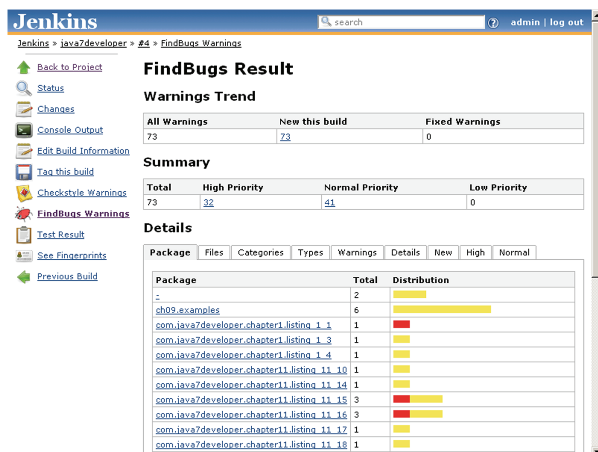


图12-15　FindBugs报告页

Jenkins、Maven和代码指标这一节到此就完成了。这一领域的工具化程度相当高（Scala和Groovy还需要更多支持），启动和运行也非常容易。如果你是CI迷，想探索Jenkins的完整能力，我们强烈推荐John Ferguson Smart不断更新的*Jenkins: The Definitive Guide*（O'Reilly）。你可能已经注意到了，与JVM多语言编程相关的构建和CI的拼图还缺了一片——我们还没处理过Clojure项目。好在Clojure社区已经专门针对纯粹的Clojure项目制作了几个构建工具，并已得到了广泛应用。其中最流行的就是Leiningen，一个专为Clojure写的构建工具。

12.6　Leiningen

要成为对开发人员有用的构建工具，关键是要具备下面这几种能力：

- 依赖管理；
- 编译；
- 测试自动化；
- 部署打包。

Leiningen对此采取的策略是分而治之。它重用已有的Java技术实现了每种功能，但却没有把所有功能都放在一个包里。

这听上去可能比较复杂，还有点恐怖，但开发人员并不会受到这种复杂性的影响。实际上，甚至没用过底层Java工具的开发人员也能使用Leiningen。我们一开始先通过一个非常简单的过程来安装Leiningen。然后讨论Leiningen的组件和整体架构，最后用Hello World项目来小试牛刀。

你将看到如何开始一个新项目，添加依赖项，使用Leiningen提供的Clojure REPL内部依赖项。这自然会让我们转而讨论如何用Leiningen在Clojure内做TDD。作为本章的收尾，我们会看一下如何打包代码，产生一个应用程序部署或供人调用的类库。

我们来看看如何开始使用Leiningen吧。

12.6.1　Leiningen入门

Leiningen非常容易上手。对于类Unix系统（包括Linux和Mac OS X），开发人员可以从掌握lein脚本开始。在GitHub上可以找到它Leiningen（在https://github.com/页面中或用自己喜欢的搜索引擎搜索Leiningen）。

把lein脚本放到PATH中并设为可执行文件后，它就可以运行了。在第一次运行lein时，它会检查需要安装哪些依赖项（还有哪些已经装上了）。只要需要，它甚至会把其他不属于Leiningen核心部分的组件也给装上。因为要安装依赖项，首次运行可能比后续运行稍慢一些。

在下一节，我们会介绍Leiningen的架构，以及为它提供核心功能的Java技术。

> **在Windows上安装Leiningen**
>
> 从一个Unix老黑客的角度来看，Windows的烦人之处是它没有为钟爱命令行的人提供赖以生存的、标准的、简单的工具。比如说，基本的Windows安装中没有通过HTTP下载文件的curl或wget工具（Leiningen需要用它们从Maven资源库中下载jar）。解决办法是用Leiningen Windows安装——带有lein.bat文件和预置的wget.ext压缩文件，为了让自行安装的lein正确工作，需要把它们放到Windows的PATH中的目录下。

12.6.2　Leiningen的架构

我们说过，Leiningen封装了一些主流的Java技术并做了简化。它封装的主要组件是Maven（版本2）、Ant和`javac`。

如图12-16所示，Maven用来做依赖项解析和管理，`javac`和Ant用来构建、运行测试和完成构建过程中的其他工作。

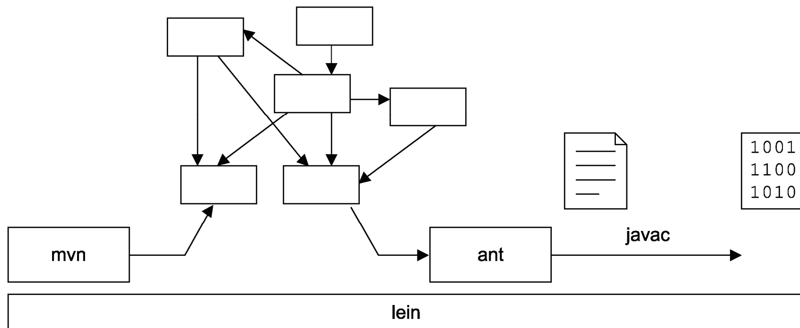


图12-16　Leiningen及其组件

高级用户可以穿过抽象层，直接使用Leiningen的底层工具。但Leiningen的基本语法和应用非常简单，不需要使用者具备使用任何底层工具的经验。

我们来看一个简单的例子，看看project.clj文件如何工作，以及在Leiningen项目生命周期中如何使用那些基本的命令。

12.6.3　Hello Lein

把lein放在PATH上之后，我们可以用它的new命令开始一个新项目：

```
ariel:projects boxcat$ lein new hello-lein
Created new project in: /Users/boxcat/projects/hello-lein
ariel:projects boxcat$ cd hello-lein/
ariel:hello-lein boxcat$ ls
README  project.clj  src  test
```

这个命令创建了一个叫做hello-lein的项目。它有项目目录，里面有个简单的描述文件README、一个project.clj文件（马上就会详细讨论），还有并列的src和test目录。

如果你把Leiningen刚刚创建的项目导入Eclipse中（比如用CounterClockwise插件），项目的布局应该如图12-17所示。

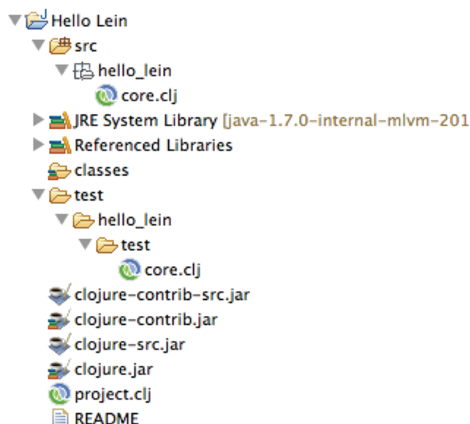


图12-17　新创建的Leiningen项目

这个项目结构是直接从Java项目上照搬过来的：有带有core.clj文件的并列test和src结构（分别用于测试和顶层代码）。另外一个重要的文件是project.clj，Leiningen用它来控制构建、保存元数据。

我们来看一下lein的new命令生成的骨架文件。

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]])
```

这个Clojure形式解析起来相当直白：有一个(defproject)的宏负责制作表示Leiningen项目的新值。这个宏需要知道项目名称（在这里是hello-lein），还需要知道项目的版本（默认是1.0.0-SNAPSHOT，12.3.1节讨论过的Maven版本号），然后是描述项目的元数据映射。

lein自带了两个元数据：一个描述字符串和一个依赖项向量，后者对于添加新的依赖项很方便。我们现在就来加一个clj-time类库。这个类库为Clojure提供Java日期和时间类库（Joda-Time，但在这个例子中你没必要知道这个Java类库）的访问接口。加上新的依赖项后，project.clj看起来应该是这样的：

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
                 [clj-time "0.3.0"]])
```

向量的第二个元素描述了要用的新依赖项类库版本。如果Leiningen在本地依赖项资源库中找不到它，会按这个版本从外部资源库获取该依赖项。

Leiningen默认从位于http://clojars.org/的资源库获取缺失的类库。因为Leiningen底层用的是Maven，所以这本质上就是一个Maven资源库。Clojars提供了一个搜索工具，可以在你知道所需类库但不知道具体版本号时提供帮助。

在这个新的依赖项就位后，你需要更新本地构建环境，可以执行`lein deps`命令。

```
ariel:hello-lein boxcat$ lein deps
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from central
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from clojure
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.pom from clojars
Transferring 2K from clojars
Downloading: joda-time/joda-time/1.6/joda-time-1.6.pom from clojure
Downloading: joda-time/joda-time/1.6/joda-time-1.6.pom from clojars
Transferring 5K from clojars
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from central
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from clojure
Downloading: clj-time/clj-time/0.3.0/clj-time-0.3.0.jar from clojars
Transferring 7K from clojars
Downloading: joda-time/joda-time/1.6/joda-time-1.6.jar from clojure
Downloading: joda-time/joda-time/1.6/joda-time-1.6.jar from clojars
Transferring 522K from clojars
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
ariel:hello-lein boxcat$
```

Leiningen已经用Maven下载了Clojure的接口，还有底层的Joda-Time JAR。我们在代码中用一下它，展示在依赖项存在的情况下如何用Leiningen作为REPL进行开发。

需要把主要源文件src/hello_lein/core.clj改成下面这样：

```
(ns hello-lein.core)

(use '[clj-time.core :only (date-time)])

(defn isodate-to-millis-since-epoch [x]
  (.getMillis (apply date-time
➥  (map #(Integer/parseInt %) (.split x "-")))))
```

它提供了一个Clojure函数，将ISO标准日期（格式为YYYY-MM-DD）转换成自Unix纪元（1970年）以来的毫秒数。

我们用Leiningen的REPL风格测试一下。先在project.clj文件中加上一行，改成下面这样：

```
(defproject hello-lein "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
                 [clj-time "0.3.0"]]
  :repl-init hello-lein.core)
```

加上这一行后，可以启动一个能访问所有依赖项的REPL，并且它已经把命名空间`hello-lein.core`中的函数引入了作用域：

```
ariel:hello-lein boxcat$ lein repl
REPL started; server listening on localhost:10886.

hello-lein.core=> (isodate-to-millis-since-epoch "1970-01-02")
86400000
hello-lein.core=>
```

这是以天为单位的日期的正确毫秒数，并且它阐明了在真实项目中使用REPL的核心原则。我们在这上面稍微展开一点，再看一个使用Leiningen REPL面向测试的工作方式。

12.6.4 用Leiningen做面向REPL的TDD

任何优秀的TDD方法，其核心都应该是一个用来开发新功能的简单基本的循环。具体到Clojure和Leiningen，其基本循环应该如下所示：

(1) 添加任何所需的新依赖项（并重新运行`lein deps`）；

(2) 启动REPL（`lein repl`）；

(3) 草拟一个新函数，并把它放到REPL的作用域中来；

(4) 在REPL内测试这个函数；

(5) 重复步骤3和4，直到该函数表现正确；

(6) 把该函数的最终版加到恰当的.clj文件上；

(7) 把刚才运行的测试用例加到测试集.clj文件中；

(8) 重启REPL，再次从第三步开始（或者第一步，如果需要新的依赖项）。

这是测试驱动开发的风格，但却避开了先写测试还是先写代码的问题，用REPL风格的TDD，这两件事是同时进行的。

之所以要在添加新函数时重启REPL（第八步），是为了能干净地编译新函数。创建新函数时，有时为了支持它，会对其他函数或环境做轻微的修改。而这些修改在把函数加入最终的源码库时很容易被忘掉。重启REPL能帮我们尽早记起这些被忘掉的修改。

这个过程清晰而简单，但还有个问题我们没提到，无论是在这里还是第11章讨论TDD时，我们都没讨论过怎么编写Clojure测试。好在这非常简单。我们来看一下用`lein new`创建新项目时它所提供的模板：


```
(ns hello-lein.test.core
  (:use [hello-lein.core])
  (:use [clojure.test]))

(deftest replace-me ;; FIXME: write
  (is false "No tests have been written."))
```

我们就用`lein test`命令来测试这个自动生成的用例，看看会发生什么（实际上你应该能猜出来）。

```
ariel:hello-lein boxcat$ lein test
Testing hello-lein.test.core
FAIL in (replace-me) (core.clj:6)
No tests have been written.
expected: false
  actual: false
Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
```

如你所见，自动生成的测试用例失败了，并且它絮叨着让你写些测试用例。那就写吧，在test文件夹里写个core.clj文件：

这个测试非常简单：使用了`(deftest)`宏，给测试命名为`(one-day)`，并且有一个跟断言语句非常相似的形式。Clojure代码的结构使得`(is)`形式读起来非常自然——几乎就像DSL一样。这个测试可以读作"自1970年1月2日以来的毫秒数等于86 400 000对吗？"我们来看一下这个测试的实际效果：

```
(ns hello-lein.test.core
  (:use [hello-lein.core])
  (:use [clojure.test]))

(deftest one-day
  (is true
    (= 86400000 (isodate-to-millis-since-epoch "1970-01-02"))))
```

这里的关键包是`clojure.test`，它提供了一些在更复杂的环境或需要用到测试固件时用来构建测试用例的形式。如果想了解得更深入，请参考Amit Rathore写的*Clojure in Action*（Manning, 2011），其中对Clojure中的TDD有全面的论述。

```
ariel:hello-lein boxcat$ lein test
Testing hello-lein.test.core
Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

在面向REPL的TDD流程就绪后，现在可以用Clojure构建一个具有相当规模且能用的应用程序了。但你终归要做一些需要跟人分享的东西。好在Leiningen有一些命令可以让你很容易地进行打包和部署。

12.6.5　用Leiningen打包和部署

Leiningen主要提供了两种代码分发办法。这两种办法本质上的区别是带不带依赖项。对应的命令分别是`lein jar`和`lein uberjar`。

我们先看一下`lein jar`：

```
ariel:hello-lein boxcat$ lein jar
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
Created /Users/boxcat/projects/hello-lein/hello-lein-1.0.0-SNAPSHOT.jar
```

下面这些是被打包进JAR文件中的东西：

```
ariel:hello-lein boxcat$ jar tvf hello-lein-1.0.0-SNAPSHOT.jar
    72 Sat Jul 16 13:38:00 BST 2011 META-INF/MANIFEST.MF
  1424 Sat Jul 16 13:38:00 BST 2011 META-INF/maven/hello-lein/hello-lein/
     pom.xml
   105 Sat Jul 16 13:38:00 BST 2011
META-INF/maven/hello-lein/hello-lein/pom.properties
   196 Fri Jul 15 21:52:12 BST 2011 project.clj
   238 Fri Jul 15 21:40:06 BST 2011 hello_lein/core.clj
ariel:hello-lein boxcat$
```

其中最明显的就是Leiningen的基本命令把Clojure源文件，而不是编译后的.class文件发出去了。这是Lisp代码的传统，因为系统的读时组件和宏会因为要处理编译后的代码而受到阻碍。

现在，我们来看看用`lein uberjar`会发生什么。它所产生的JAR不仅包含代码，还有依赖项。

```
ariel:hello-lein boxcat$ lein uberjar
Cleaning up.
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
Copying 4 files to /Users/boxcat/projects/hello-lein/lib
Created /Users/boxcat/projects/hello-lein/hello-lein-1.0.0-SNAPSHOT.jar
Including hello-lein-1.0.0-SNAPSHOT.jar
Including clj-time-0.3.0.jar
Including clojure-1.2.1.jar
Including clojure-contrib-1.2.0.jar
Including joda-time-1.6.jar
Created /Users/boxcat/projects/hello-lein/
➥ hello-lein-1.0.0-SNAPSHOT-standalone.jar
```

看到了吧，这个JAR中不仅有代码，还有依赖项，以及依赖项的依赖项，这称为依赖的传递闭包图。也就是说它是一个可以完全独立运行的包。

当然，因为所有依赖项都打包了，所以这也意味着`lein uberjar`打包的文件要比`lein jar`的文件大很多。即便是我们这个简单的小例子，其差异也相当鲜明：

```
ariel:hello-lein boxcat$ ls -lh h*.jar
-rw-r--r--  1 boxcat  staff   4.1M 16 Jul 13:46
hello-lein-1.0.0-SNAPSHOT-standalone.jar
-rw-r--r--  1 boxcat  staff   1.7K 16 Jul 13:46
hello-lein-1.0.0-SNAPSHOT.jar
```

你可以这样理解`lein jar`和`lein uberjar`：如果要构建一个类库（构建在其他类库之上），或者要将它作为依赖项，就用`lein jar`。如果是构建一个最终用户使用的Clojure应用程序，而不是交给用户去扩展的工件，就用`lein uberjar`。

你已经见过用Leiningen如何开始、管理、构建和部署Clojure项目了。Leiningen还有很多内置的实用命令，还有一个强大的插件系统让你可以对它进行定制。想要对Leiningen能做什么有更多了解，只要调用时不带命令就可以了，单用`lein`。

我们在下一章构建Clojure的Web应用时还会遇到Leiningen。

12.7　小结

有优秀Java开发人员参与的项目肯定有快速、可重复、简单的构建。如果软件不能快速稳定地构建，就会浪费大量的时间和金钱，包括你自己的！

理解编译—测试—打包这一基本构建周期是确立良好构建流程的关键。毕竟，你不能测试还没编译的代码！

Maven将构建周期的概念发扬光大，扩展为在所有Maven项目中都能保持一致的项目周期。这种惯例优先（于配置）的方式对于大型软件团队非常有帮助，但有些项目可能需要更多的灵活性。

Maven还解决了依赖管理的问题，因为几乎所有项目都要依赖第三方类库，这个难题一直困扰着Java和开源世界。

把构建流程挂到CI环境中，开发人员就能得到迅捷无比的反馈，还可以毫无畏惧地快速合并修改。

Jenkins是一个流行的CI服务器，不仅能构建几乎所有类型的项目，还能通过它庞大的插件系统提供丰富的报告。假以时日，开发团队就能让Jenkins执行各种构建，覆盖范围可以从快速单元测试构建到系统集成构建。

Leiningen是Clojure项目的自然之选。它用一个非常清爽的构建和部署工具，把紧凑的TDD循环和REPL方式结合在了一起。

我们接下来会讨论快速Web开发，自从第一个基于Java的Web框架出现以来，大多数优秀的Java开发人员都曾为这一主题奋斗过。

第 13 章 快速Web开发

本章内容

- 为什么Java不是快速Web开发的理想选择
- Web框架的选择标准
- 基于JVM的Web框架比较
- 认识Grails（与Groovy）
- 认识Compojure（与Clojure）

快速Web开发很重要，非常重要。在全球的商业和社交活动中，数量庞大的网站和由Web技术驱动的应用程序占据着主导地位。企业（特别是创业公司）的生死取决于他们向市场投放新产品或新特性的速度。如今的终端用户希望新功能的出现和bug的消失能像变魔术一样快，他们越来越没耐心了。

可大多数Java上的Web框架在支持快速Web开发的能力上都有限，为了不在激烈的竞争中死掉，很多组织都纷纷转向PHP和Rails之类的技术。

作为一个优秀的Java开发人员，你该何去何从？好在最近JVM上出现了动态层语言，现在JVM上也有快速Web开发的理想选择。Grails（Groovy）和Compojure（Clojure）就是这样的框架，它们能满足你要求的快速Web开发能力。也就是说你不用放弃强大而又灵活的JVM，在跟PHP和Rails这样的技术竞争时也不用比它们多花几个小时了。

Java EE 6：Java的快速Web开发是否向前迈进了一步？

相比J2EE（曾因JSP、Servlet和EJB API饱受诟病），Java企业版（Java EE）6已经有了长足的发展。尽管Java EE 6所做的改进（在JSP、Servlet和EJB API上有明显体现）仍然受限于Java静态类型系统和编译方面的问题。

本章一开始会解释一下为什么Java上的Web框架不是快速Web开发的理想选择。顺着这个解释，你会了解优秀的Web框架应该满足哪些标准。通过一些定量的研究，以及Matt Raible的工作，你会理解如何用Web框架的20条标准对各种JVM Web框架进行评级。

Grails是快速Web开发框架的领导者之一，它满足了其中的很多标准。我们会带你过一遍这个基于Groovy的Web框架，炙手可热的Rails框架对它产生了很大的影响。

我们还会讨论作为Grails备选的Compojure，它是一个基于Clojure的Web框架，可以实现非常精炼的Web编程和快速开发。

让我们先来看看为什么基于Java的Web框架不一定是现代Web项目的理想选择。

13.1　Java Web 框架的问题

第7章讨论过多语言编程金字塔和编程语言的三个层次，我们在图13-1中再次重复一下。

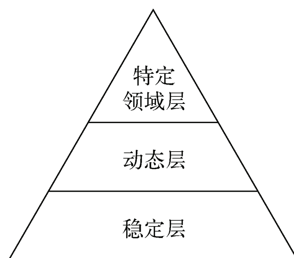


图13-1　多语言编程金字塔

Java位于稳定层，所以它的所有Web框架也属于这一层。作为一门流行而又成熟的语言，Java中有很多种Web框架，比如：

- Spring MVC
- GWT
- Struts 2
- Wicket
- Tapestry
- JSF（及其他与Faces相关的类库）
- Vaadin
- Play
- 以前那些普通的JSP/Servlet

在这一领域，Java没有公认的领头羊，并且源于Java的这一分支根本就不是快速Web开发的理想选择。Struts 2曾经流行一时，该项目的前任领导者说过这样一段话：

> 我堕落了:–)，我现在更喜欢用Rails——因为前面提到的简洁性，因为我再也不用“构建”和“部署”了。兄弟姐妹们，我要提醒你们，如果你们想吸引Rails开发人员……或者要避免像我这样的“背叛Java的Web开发人员”流失:–)，这类事情是你们必须克服的障碍。
>
> ——Craig McClanahan，2007年10月23日
>
> （http://markmail.org/thread/qfb5sekad33eobh2）

本节会谈到Java为什么不是快速Web开发的理想选择。我们先来看看编译型语言为什么会在开发Web应用时拖后腿。

13.1.1 Java编译为什么不好

Java是编译型语言，这就是说你每次修改代码，都必须重复下面的步骤：

- 重新编译Java代码；
- 停止Web服务器；
- 把改过的应用重新部署到Web服务器中；
- 启动Web服务器。

这会浪费大量的时间！特别是有很多小修改时，比如修改控制权的目标或界面的微调。

注意 应用服务器和Web服务器之间的界限已经变得模糊了。这是因为JEE 6的出现（可以在Web容器内运行EJB），也因为大多数应用服务器都是高度模块化的。我们提到的“Web服务器”是指任何有Servlet容器的服务器。

如果你是一位经验丰富的Web开发人员，应该知道有些技术可以解决这个问题。其中大多数都是不用停止和启动Web服务器就能应用代码修改，也被称为*热部署*。热部署或者把全部资源（比如整个WAR文件）都换掉，或者只选其中几个（比如单个JSP页面）资源进行替换。但热部署不是100%可靠（因为类加载的限制和容器中的bug），并且大多数情况下，Web服务器仍然需要执行昂贵的代码重编译操作。

用JRebel和LiveRebel执行热部署

如果你必须要用Java Web框架，我们强烈向你推荐JRebel和LiveRebel（http://www.zeroturnaround.com/jrebel/）。JRebel确实具备一些惊艳的JVM技巧，它处在IDE和Web服务器之间，源码发生变化时能自动将这些变化反应到正在运行的Web服务器上（LiveRebel用于生产环境的部署）。这种热部署基本没什么问题，并且这些工具实际上是解决热部署问题的行业标准。

一般来说，Java Web框架为代码修改而产生的周转时间太长。但这不是唯一的问题，另外一个拖慢Web开发速度的不利因素是语言的灵活性，这是静态类型系统的弱项。

13.1.2 静态类型为什么不好

在开发新产品或新功能的早期阶段，保持用户展示层设计的开放性（对于类型而言）通常都是明智之举。对于用户来说，要求数值精确到小数位，或让书单变成图书和玩具混合的清单都是非常合理的要求。静态类型可能会成为这类需求的巨大障碍。如果你必须把一个`Book`对象列表

变成`BookOrToy`[①]对象的列表，则只能在代码中把这个静态类型全改掉。

尽管在容器类里能用基本类型（比如Java的`Object`类）作为对象的类型，但这肯定不是最佳实践——这简直是一下子回到了泛型。

因此，选择基于动态层语言编写的Web框架无疑是个有效选项。

注意　Scala当然是静态类型语言。但由于其先进的类型推断能力，它能规避很多由Java静态类型实现方式引发的问题。也就是说，Scala可以是、也确实是可用的Web层语言。

在你继续深入研究和选择Web开发的动态语言之前，我们先后退一步，让视野更开阔一点。想一想优秀的快速Web开发框架应该符合哪些标准。

13.2　选择 Web 框架的标准

Java这么多年的顶级编程语言地位不是白给的，Java中可供选择的Web框架有很多。最近基于其他JVM语言（比如Groovy、Scala和Clojure）的Web框架也雄起了。但不幸的是，这么多年来还没有一个能在这一领域确立自己的霸主地位，选哪个框架完全看个人偏好。

Web框架应该能提供大量帮助。必须能用一些标准进行评估，它能满足的标准越多，开发Web应用的速度可能会越快。

Matt Raible[②]总结出了评判Web框架的20条标准[③]。表13-1对这些标准作了简要介绍。

表13-1　Web框架的20条标准

标　　准	示　　例
开发人员的工作效率	能用1天或5天搭出一个CRUD页面吗
开发人员的看法	用起来有意思吗
学习曲线	学了一个礼拜或一个月后能干活吗
项目健康状况	项目陷入绝境了吗
开发人员的充足性	能找到经验丰富的开发人员吗
就业趋势	将来能招到人吗
模板化	遵循DRY（不重复自己）原则吗
组件	自带日期选择器之类的东西吗

① 小心！如果你的域对象中有Or或And这样的字眼出现，那你很可能违反了我们在第11章讨论的SOLID原则。

② AppFuse（http://appfuse.org）的作者，AppFuse是一个Web开发基础平台，它集成了Java中各种流行的Web框架，并提供了所有Web系统开发过程中都需要开发的一些功能，比如登录、用户密码加密、用户管理、为不同的用户展现不同的菜单。它可以自动生成40%~60%的代码，还自带了一些默认的CSS样式，使用这些样式能快速改变整个系统的外观，还具备自动测试的能力。——译者注

③ Matt Raible, “Comparing JVM Web Frameworks” （March 2011）, presentation. http://raibledesigns.com/rd/page/publications。

（续）

标　　准	示　　例
Ajax	支持客户端的异步Javascript调用吗
插件或附加项	能加上Facebook集成之类的功能吗
扩展性	默认的控制器处理的并发用户数能到500+吗
测试支持	能做测试驱动的开发吗
I18n和l10n	自带其他语种和地域的支持吗
校验	能轻松校验用户输入并迅速反馈吗
多语言支持	能同时用（比如说）Java和Groovy吗
文档/教程的质量	常见的用例和问题在文档中都有体现吗
出版图书	有没有行业专家用过它，并分享了自己的战斗事迹
REST支持（服务器端和客户端）	它能按HTTP的设计宗旨使用该协议吗
移动支持	是否很容易就能支持Android、iOS和其他移动设备
风险程度	是用来做“保存食谱”的应用程序或是“核电站控制器”

你看到了，这个清单很长，在做决定时，你需要想好各个标准的权重。不过Matt很勇敢[①]，他最近在这一领域做了一些研究，尽管其研究结果引发了激烈的争论，但真相总算是开始浮现了。如果给那些对快速Web开发最重要的标准赋予较高的权重，各种框架的得分（总分为100）如图13-2所示。这些标准应该是：开发人员的工作效率、测试支持和文档的质量。

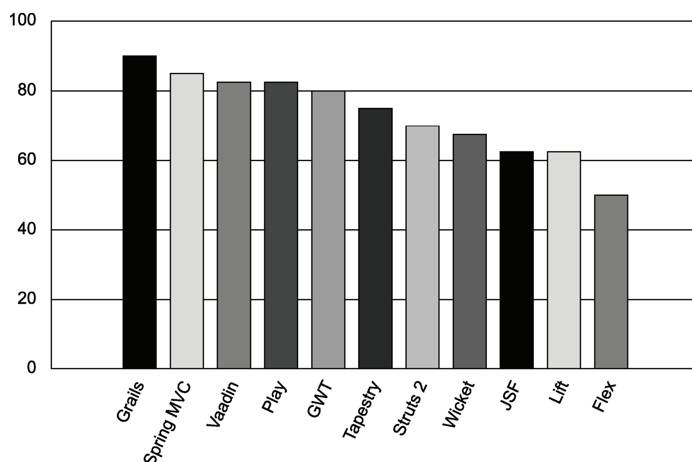


图13-2　Matt Raible对JVM框架的加权评级

不同的人需求可能会不同，在http://bit.ly/jvm-frameworks-matrix上可以很容易地修改Matt的权重，运行自己的分析，产生自己的图形。

① 你应该能想象得到，人们对于自己喜爱的Web框架有多大的热情！

提示　在你选定框架之前，我们强烈建议你在两到三个框架上按自己的标准做一些功能原型。

现在你知道该用哪些标准进行评估了，并且还能利用Matt提供的工具，所以在选择快速Web开发框架时，你可以做出明智的选择。在我们的加权标准分析中脱颖而出的是Grails框架（Compojure没有名列前茅，但因为它还非常年轻，所以我们预计它在不久的将来能迅速蹿升到领导阵营中）。

我们来看看获胜者Grails！

13.3　Grails 入门

Grails是基于Groovy的快速Web应用框架，它集成了多个第三方类库，包括Spring、Hibernate、JUnit和Tomcat服务器等。它是一个完备的Web框架，13.2节列出的20条标准它全都满足。还有一点很重要，Grails在很大程度上借鉴了Rails中惯例优先的原则。如果能依照惯例编码，框架会帮你做很多套路化的工作。

我们在这一节中会讨论如何搭建你的第一个快速启动应用。在搭建快速启动应用的过程中，你会看到很多可以证明Grails“快速”的证据。我们还会指出Grails中那些需要进一步探索的重要技术，从而让你可以构建出能用于生产环境的、正儿八经的应用程序。

不喜欢Groovy？试试Spring Roo

Spring Roo（www.springsource.org/roo）是跟Grails基于同样原则开发的快速Web开发框架，但它的核心语言是Java，并且向开发者开放了更多的Spring DI框架。我们觉得它没Grails成熟，但如果你确实不喜欢Groovy，这也是个不错的备选。

如果不熟悉Groovy，可能需要认真温习一下第8章。在你能跟Groovy融洽相处后，请下载Grails并安装。附录C中有该过程的完整指导。

装好Grails之后，就该开始你的第一个Grails项目了！

13.4　Grails 快速启动项目

这一节会介绍一个Grails快速启动项目，重点展示Grails作为快速Web框架的亮点。用Grails创建Web应用所需的步骤如下：

- 创建域对象；
- 测试驱动开发；
- 域对象的持久化；
- 创建测试数据；
- 控制器；

- ❑ GSP视图；
- ❑ 脚手架和自动化的UI创建；
- ❑ 快速开发的周转时间。

说得具体点，我们准备搞一个角色扮演游戏[①]中的基本构件（`PlayerCharacter`）。到本节结束的时候，你会创建一个具备以下能力的简单的域对象（`PlayerCharacter`）：

- ❑ 进行一些运行时测试；
- ❑ 预先准备好测试数据；
- ❑ 可以保存到数据库中；
- ❑ 具有可以进行CRUD操作的基本UI。

Grails节省时间的第一个法宝就是自动创建好项目结构。运行`grails create-app <my-project>`命令，马上就能得到一个可以构建的项目！你需要做的唯一一件事情就是保证能接入互联网，因为它要下载标准的Grails依赖项（比如Spring、Hibernate、JUnit、Tomcat服务器等）。

Grails用来管理和下载依赖项的工具是Apache Ivy。它下载和管理依赖项的概念跟第12章介绍的Maven非常像。下面这个命令会创建一个叫做pcgen_grails的应用程序，包括一个依照Grails的传统优化过的项目结构。

```
grails create-app pcgen_grails
```

依赖项下载完成，其他自动安装步骤也完成之后，你应该就会得到一个如图13-3所示的项目结构。

```
pcgen_grails
  application.properties     ---> basic application info/versioning
  + grails-app
    + conf                   ---> location of configuration artifacts
      + hibernate            ---> optional hibernate configuration
      + spring               ---> optional spring configuration
    + controllers            ---> location of controller artifacts
    + domain                 ---> location of domain classes
    + i18n                   ---> location of message bundles for i18n
    + services               ---> location of services
    + taglib                 ---> location of tag libraries
    + util                   ---> location of special utility classes
    + views                  ---> location of views
      + layouts              ---> location of layouts
  + lib
  + scripts                  ---> scripts
  + src
    + groovy                 ---> optional; location for Groovy source files
                                  (of types other than those in grails-app/*)
    + java                   ---> optional; location for Java source files
  + test                     ---> generated test classes
  + web-app
    + WEB-INF
```

图13-3 Grails项目的布局

有了项目结构就可以开始生产一些能运行的代码了！首先要创建域对象类。

① 想想《龙与地下城》或《指环王》。

13.4.1　创建域对象

Grails以域对象为应用程序的核心，因此鼓励你按域驱动设计（Domain-Driven Design，DDD）的方式来考虑问题[①]。执行`grails create-domain-class`命令可以创建域对象。

下面的例子创建了一个`PlayerCharacter`类，用来表示游戏中的角色：

```
cd pcgen_grails
grails create-domain-class com.java7developer.chapter13.PlayerCharacter
```

Grails会自动为你创建下面的文件：

- 一个表示域对象的PlayerCharacter.groovy源文件（在目录grails-app/domain/com/java7developer/chapter13下）；
- 开发单元测试用的PlayerCharacterTests.groovy源文件（在目录test/unit/com/java7developer/chapter13下）。

看，Grails在鼓励你写单元测试！

还需要给`PlayerCharacter`定义一些属性，比如`strength`、`dexterity`和`charisma`。有了这些属性，你就可以开始构想游戏中的角色如何跟想象的世界交互[②]。但刚刚看过第11章，你当然想先写测试！

13.4.2　测试驱动开发

按TDD的方式，我们要先写个失败测试，然后实现`PlayerCharacter`让测试通过。

我们还准备利用Grails的域对象自动校验特性。在Grails中，可以自动在任何域对象上调用`validate()`方法，以确保该对象的有效性。代码清单13-1会测试`strength`、`dexterity`和`charisma`三项统计量都是3到18之间的数值。

代码清单13-1　`PlayerCharacter`的单元测试

```
package com.java7developer.chapter13

import grails.test.*

class PlayerCharacterTests extends GrailsUnitTestCase {    ❶ 扩展 Grails Unit-TestCase

  PlayerCharacter pc;

  protected void setUp() {
    super.setUp()
    mockForConstraintsTests(PlayerCharacter)    ❷ 注入Validate()
  }

  protected void tearDown() {
    super.tearDown()
  }
```

① 想了解DDD（由Eric Evans提出）的更多内容，请访问域驱动设计社区（http://domaindrivendesign.org/）。

② Gweneth是不是应该善于摔跤、杂耍，或面带微笑地解除对手的武装？

```
  void testConstructorSucceedsWithValidAttributes {
    pc = new PlayerCharacter(3, 5, 18)
    assert pc.validate()                          ❸ 通过校验
  }

  void testConstructorFailsWithSomeBadAttributes() {
    pc = new PlayerCharacter(10, 19, 21)
    assertFalse pc.validate()                     ❹ 校验失败
  }
}
```

Grails的单元测试都应该扩展自`GrailsUnitTestCase`❶。跟所有标准的JUnit测试一样，它也有`setUp()`和`tearDown()`方法。但为了在单元测试阶段用Grails内置的`validate()`方法，必须通过`mockForConstraintsTest`方法把它拉进来❷。这是因为Grails把`validate()`当做集成测试的关注点，通常只有这样才能用它。但如果想要更快地得到反馈，可以把它放到单元测试阶段。接下来，可以调用`validate()`来检查域对象是否有效❸❹。

现在可以执行下面的命令来运行测试了：

```
grails test-app
```

这个命令既运行单元测试也会运行集成测试（不过我们现在只有单元测试），并且从控制台中的输出来看，测试失败了。

要了解测试失败的原因，需要到target/test-reports/plain目录下去找。对于这个程序，要找到TEST-unit-unit-com.java7developer.chapter13.PlayerCharacterTests.txt文件。这个文件会告诉你测试失败是因为在尝试创建新的`PlayerCharacter`时，没找到匹配的构造方法。这很容易理解，因为`PlayerCharacter`域对象还什么都没有呢！

现在你可以把`PlayerCharacter`搭起来，重复运行测试直到通过。按你的想法加上`strength`、`dexterity`和`charisma`三个属性。但为了在这些属性上设定`minimum (3)`和`maximum(18)`的限制，需要用特殊的限定语法。那样就可以用Grails提供的默认`validate()`方法了。

Grails中的限定

Grails中的限定是在Spring validator API基础上实现的。可以用它们指定域类型属性的校验需求。Grails的限定很多（在代码清单13-2中用到了`min`和`max`），你还可以根据需要自行编写限定。参见http://grails.org/doc/latest/guide/validation.html了解详情。

下面这段代码中的`PlayerCharacter`类中仅包含了让它可以通过测试的最基本的属性和限定。

代码清单13-2 `PlayerCharacter`类

```
package com.java7developer.chapter13

class PlayerCharacter {

  Integer strength
  Integer dexterity                               ❶ 要持久化的类型变量
  Integer charisma

  PlayerCharacter() {}
```

```
  PlayerCharacter(Integer str, Integer dex, Integer cha) {
    strength = str
    dexterity = dex
    charisma = cha
  }

  static constraints = {
    strength(min:3, max:18)
    dexterity(min:3, max:18)
    charisma(min:3, max:18)
  }
}
```

❷ 可以通过测试的构造方法

❸ 用于校验的限定

PlayerCharacter类相当简单。有三个会自动保存到PlayerCharacter表中的基本属性❶。有一个带三个参数的构造方法❷。那个特殊的static代码块确定了validate()方法要检查的min和max值❸。

PlayerCharacter类变具体后，测试应该能很痛快地通过了（再次运行grails test-app）。如果遵循TDD方式，到这个阶段就该着手重构PlayerCharacter和测试了，以便让代码更加清爽。

Grails还会确保域对象保存到数据存储中。

13.4.3　域对象持久化

持久化是由Grails自动处理的，因为Grails认为类中所有具有明确类型的域变量都应该保存到数据库中。Grails会自动把域对象映射到同名的表中。对于PlayerCharacter域对象而言，三个属性（strength、dexterity和charisma）全部是Integer类型，所以都会映射到PlayerCharacter表中。Grails默认使用Hibernate，并会提供一个HSQLDB内存数据库（我们在第11章提到过它，那时用做伪装测试替身），但你可以用自己的数据源取代默认数据源。

grails-app/conf/DataSource.groovy文件里是数据源的配置。可以在这里为每种环境设定数据源。记住，Grails已经在pcgen_grails里给出了默认使用HSQLDB的实现，所以无需任何修改就可以运行它。但代码清单13-3中给出了使用其他数据库的配置供参照。

代码清单13-3　可能的pcgen_grails数据源

```
dataSource {}

environments {
    development { dataSource {} }
    test { dataSource {} }

    production {
        dataSource {
            dbCreate = "update"
            driverClassName = "com.mysql.jdbc.Driver"
            url = "jdbc:mysql://localhost/my_app"
            username = "root"
            password = ""
        }
    }
}
```

生产数据源

数据库驱动

JDBC连接URL

比如说，可以在生产环境中使用MySQL数据库，而开发和测试环境中还用HSQLDB。这些都是相当标准的Java数据库连接（JDBC）配置，你对它们应该已经很熟悉了。

Grails开发者也考虑到了手工创建测试数据的问题，所以他们提供了一种机制，可以在应用启动时将数据预填充到数据库中。

13.4.4 创建测试数据

测试数据的创建通常是由Grails的`BootStrap`类完成的，它在grails-app/conf/BootStrap.groovy中。只要Grails应用或Servlet容器启动，就会运行`init`方法。这和大多数Java Web框架用的启动servlet所起的作用是一样的。

注意 可以用`Bootstrap`类做所有初始化操作，但现在我们主要讨论测试数据。

代码清单13-4在初始化阶段生成了两个`PlayerCharacter`域对象，并把它们存到了数据库里。

代码清单13-4 为`pcgen_grails`引导测试数据

```
import com.java7developer.chapter13.PlayerCharacter

class BootStrap {

  def init = { servletContext ->                    ❶ 在Servlet上下文启动时引导
    if (!PlayerCharacter.count()) {
      new PlayerCharacter(strength: 3, dexterity: 5, charisma: 18)
                                  .save(failOnError: true)
      new PlayerCharacter(strength: 18, dexterity: 10, charisma: 4)
                                  .save(failOnError: true)
    }
  }

  def destroy = {}
}
```

每次把代码部署到Servlet容器中都会执行`init`方法（即应用启动和Grails自动部署时）❶。为了确保不会覆盖掉任何已有数据，可以对已有的`PlayerCharacter`实例执行简单的`count()`方法。如果确定没有实例，可以创建一些。这里有个很重要的特性：如果有异常抛出，或所构造的对象无法通过校验，则可以肯定对象不会保存到数据库中。如果愿意，可以在`destroy`方法中执行清除操作。

有了一个带有存储支持的基本域对象后就可以进入下一阶段了：在Web页面上显示域对象。为此需要构建一个Grails控制器，你应该不会对这个源自MVC设计模式的术语感到陌生。

13.4.5 控制器

Grails遵循MVC设计模式，用控制器来处理来自客户端（一般是浏览器）的Web请求。Grails的惯例是给每个域对象配一个控制器。

要创建域对象`PlayerCharacter`的控制器只需要执行下面这条命令：

```
grails create-controller com.java7developer.chapter13.PlayerCharacter
```

重要的是指明域对象的完全限定类名，包括包名。

命令执行完成后应该能发现下面这些文件：

- ❑ `PlayerCharacter`域对象的控制器的PlayerCharacterController.groovy源文件（在grails-app/controller/com/java7developer/chapter13目录下）；
- ❑ 开发控制器单元测试的PlayerCharacterControllerTests.groovy源文件（在test/unit/com/java7developer/chapter13目录下）；
- ❑ grails-app/view/playerCharacter文件夹（稍后会用到）。

控制器以简单的方式支持REST风格的URL和操作映射。假设要把REST风格的URL http://localhost:8080/pcgen_grails/playerCharacter/list映射到一个返回`PlayerCharacter`对象列表的方法上。按照Grails惯例优于传统的方式可以用最少的源码把URL映射到`PlayerCharacterController`类中。这个URL是由下面这些元素组成的：

- ❑ 服务器（http://localhost:8080/）；
- ❑ 基础项目（pcgen_grails/）；
- ❑ 控制器名称的衍生部分（playerCharacter/）；
- ❑ 在控制器里声明的操作块变量（list）。

要在代码中看到这些元素，请用代码清单13-5替换已有的PlayerCharacterController.groovy源码。

代码清单13-5　`PlayerCharacterController`

```
package com.java7developer.chapter13

class PlayerCharacterController {
  List playerCharacters
  def list = {
    playerCharacters = PlayerCharacter.list()      ❶ 返回PlayerCharacter对象列表
  }
}
```

使用Grails的惯例处理方式，`playerCharacter`的属性会用在REST风格的URL指向的页面中❶。

但如果现在就启动程序，然后访问http://localhost:8080/pcgen_grails/playerCharacter/list，是不会成功的，因为还没创建JSP或GSP页面。现在我们就来解决这个问题。

13.4.6　GSP/JSP页面

用Grails既可以创建GSP页面，也可以创建JSP页面。这一节会创建一个简单的GSP页面，用来显示`PlayerCharacter`对象的列表（设计师、Web开发者和HTML/CSS大拿们，现在请移开你们的视线！）

代码清单13-6是GSP页面grails-app/view/playerCharacter/list.gsp的代码。

代码清单13-6　PlayerCharacter列表的GSP页面

```
<html>
   <body>
     <h1>PC's</h1>
     <table>
       <thead>
         <tr>
           <td>Strength</td>
           <td>Dexterity</td>
           <td>Charisma</td>
         </tr>
       </thead>
       <tbody>
       <% playerCharacters.each({ pc -> %>          ❶ 开始循环
         <tr>
           <td><%="${pc?.strength}"%></td>          ❷ 输出属性
           <td><%="${pc?.dexterity}"%></td>
           <td><%="${pc?.charisma}"%></td>
         </tr>
         <%})%>                                      ❸ 循环结束
       </thead>
     </table>
   </body>
</html>
```

HTML非常简单，关键是如何用Groovy脚本。你会注意到我们在第8章介绍的Groovy函数字面值语法，它简化了集合循环操作❶。接着是对角色属性的引用（注意安全的null解引用操作符的使用）❷，然后结束函数字面值❸。

既然域对象、控制器和它的显示页面都准备好了，接下来就可以启动Grails应用了！执行下面这条命令即可：

```
grails run-app
```

Grails会自动在http://localhost:8080上启动一个Tomcat，并把pcgen_grails应用部署上去。

警告　很多开发人员已经装过Tomcat服务器了。如果想同时启动多个Tomcat实例，就要修改端口号，端口8080只能有一个实例监听。

如果你打开浏览器访问http://localhost:8080/pcgen_grails/，会看到页面上列出了PlayerCharacterController，如图13-4所示。

点击com.java7developer.chapter13.PlayerCharacterController链接，就会进入PlayerCharacter域对象的列表页。

尽管做这个GSP页面相当快，但如果框架能帮你做岂不是更好？用Grails的脚手架功能可以迅速做出域对象CRUD页面的原型。

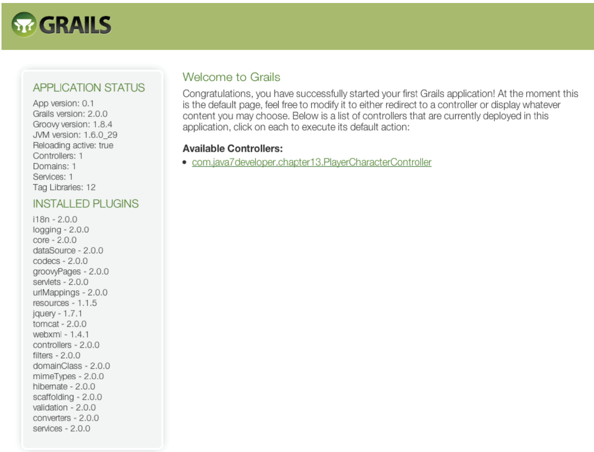


图13-4　pcgen_grails主页

13.4.7　脚手架和UI的自动化创建

Grails可以用它的脚手架（scaffolding）特性自动创建用来执行域对象CRUD操作的UI。

要使用脚手架特性，请用代码清单13-7替换PlayerCharacterController.groovy源文件中的代码：

代码清单13-7　带脚手架的PlayerCharacterController

```
package com.java7developer.chapter13

class PlayerCharacterController {
  def scaffold = PlayerCharacter          ❶ 用于PlayerCharacter的脚手架
}
```

PlayerCharacterController类非常简单。依照惯例将域对象的名称赋值给脚手架变量❶，Grails马上就可以构建默认UI。

请暂时把list.gsp改成list_original.gsp，以防它会妨碍脚手架产生相应的文件。改好之后，刷新http://localhost:8080/pcgen_grails/playerCharacter/list页面，就会看到自动生成的PlayerCharacter域对象列表，如图13-5所示。

GRAILS

Home　New PlayerCharacter

PlayerCharacter List

Strength	Dexterity	Charisma
3	5	18
18	10	4

图13-5　PlayerCharacter实例列表

在这个页面中也可以创建、更新和删除PlayerCharacter对象。请确保添加了两个PlayerCharacter域对象记录，然后进入下一节了解与代码修改的快速周转有关的内容。

13.4.8 快速周转的开发

Grails的run-app命令为快速Web开发中的“快速”贡献了一点儿特殊的东西。用Grails的run-app命令运行的应用程序，其源码会和服务器连接起来。尽管这在生产环境中不是什么明智之举（因为会影响性能），但对于开发和测试来说非常重要。

提示 对于生产环境，一般都是用grails war创建WAR文件，然后通过标准的开发流程进行部署。

如果Grails应用中的源码改了，这些变化会自动反映到服务器上[1]。我们来试试，改一下PlayerCharacter域对象：在PlayerCharacter.groovy文件中加一个变量name，存一下。

```
String name = 'Gweneth the Merciless'
```

现在刷新http://localhost:8080/pcgen_grails/playerCharacter/list页面，就能看到PlayerCharacter对象上新加了name属性这一列。注意到了吗？不用停Tomcat，不用重新编译代码，其他的什么也不用做。Grails就是靠这种几乎即时生效的速度确立了它快速Web开发框架的领导地位。

我们对快速启动项目的介绍就到此为止了，你应该体验了一把用Grails做快速Web开发。当然，还有很多可以对默认行为进行定制的方法值得探索。现在我们就去看看吧。

13.5 深入 Grails

可惜啊，短短一章的篇幅无法承载Grails框架的所有内容，因为它需要一本书！在这一节，我们再为新加入Grails阵营的开发人员讲一些值得探索的领域：

- 日志；
- GORM：Grails对象—关系映射；
- Grails插件。

另外，也可以到http://www.grails.org网站上去看看，上面有关于这些主题的基本教程。Glen Smith和Peter Ledbrook写的*Grails in Action*（Manning，2009）也值得仔细阅读。

我们从Grails的日志入手吧。

13.5.1 日志

Grails的日志功能是由log4j提供的，在grails-app/conf/Config.groovy文件中配置。

① 对于大多数源码来说都是如此，只要没改出错来就行。

比如说，你可能想要chapter13包中的代码显示WARN消息，而域对象类`PlayerCharacter`只显示ERROR消息。要满足这一要求，可以把下面这段代码放到log4j的配置文件Config.groovy文件中。

```
log4j = {
    ...
    warn   'com.java7developer.chapter13'
    error  'com.java7developer.chapter13.PlayerCharacter',
           'org.codehaus.groovy.grails.web.servlet',  //  控制器
    ...
}
```

日志配置就跟你过去用log4j的log4j.xml配置一样灵活。

接下来我们会看看Grails中的对象关系映射技术GORM。

13.5.2　GORM：对象关系映射

GORM是用Spring/Hibernate实现的，这是Java开发人员非常熟悉的技术组合。它所涵盖的功能非常广泛，但其核心功能非常像Java的JPA。

要想马上实验一下它的持久化行为，可以执行如下命令打开Grails控制台：

```
grails console
```

还记得第8章讲的Groovy控制台吗？这个Grails应用环境跟那个非常类似。

首先，我们保存一下`PlayerCharacter`域对象：

```
import com.java7developer.chapter13.PlayerCharacter
new PlayerCharacter(strength:18, dexterity:15, charisma:15).save()
```

`PlayerCharacter`保存好后有很多种办法可以读取它。最简单的办法是通过Grails添加到域对象类中的隐含`id`属性取回可写的完整实例。在控制台用下面这段代码换掉前面那段并执行。

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
assert 18 == pc.strength
```

要更新对象，修改一些属性然后再次调用`save()`方法。请再次清空控制台并运行下面这段代码。

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
pc.strength = 5
pc.save()
pc = PlayerCharacter.get(1)
assert 5 == pc.strength
```

要删除对象请用`delete()`方法。再次清空控制台并运行下面的代码，删除`PlayerCharacter`。

```
import com.java7developer.chapter13.PlayerCharacter
def pc = PlayerCharacter.get(1)
pc.delete()
```

GORM具备完整丰富的多对一、多对多关系声明能力，以及其他我们熟悉的Hibernate/JPA所支持的关系声明能力。

现在我们去看看从Rails“拿来”的插件概念。

13.5.3 Grails插件

Grails有大量插件，可以帮开发人员完成常见的Web开发任务。其中最流行的插件有：

- Cloud Foundry Integration（用于将应用部署到云服务上）；
- Quartz（用于计划调度）；
- Mail（用于处理电子邮件）；
- Twitter、Facebook（用于社交网络集成）。

要查看有哪些插件可用，请执行如下命令：

```
grails list-plugins
```

然后可以执行`grails plugin-info [名称]`查看插件的更多信息，用感兴趣的插件名称替换[名称]就可以了。此外，也可以访问http://grails.org/plugins/深入了解这些插件及其生态系统的信息。

要安装插件，请运行`grails install-plugin [名称]`，用要安装的插件名称替换[名称]。比如说，为了更好地支持日期和时间，可以安装Joda-Time插件。

```
grails install-plugin joda-time
```

装上Joda-Time插件后，可以给`PlayerCharacter`加上`LocalDate`属性。把下面的`import`语句加到域对象类中。

```
import org.joda.time.*
import org.joda.time.contrib.hibernate.*
```

把下面这个属性加到`PlayerCharacter`中。

```
LocalDate timestamp = new LocalDate()
```

为什么这跟引用JAR文件中的API不同呢？因为Joda-Time插件会确保该类型跟Grails惯例优先的原则兼容。这就是说Joda-Time的类型是映射到数据库类型上的，并且完全支持它的映射和脚手架处理。如果现在回到http://localhost:8080/pcgen_grails/playerCharacter/list页面中，会看到列出了日期。

借助插件的这类支持，Grails开发人员可以用很短的时间构建出数量惊人的功能。

我们对Grails的初次拜访结束了，但本章中快速Web开发的故事还没讲完。下一节会讨论Clojure的快速Web开发类库Compojure。熟悉Clojure的开发人员可以借助它用简洁的Clojure代码迅速构建出小到中型的Web应用。

13.6　Compojure 入门

开发Web最致命的想法就是把什么网站都当成Google来设计。对于Web应用来说，过度设计和设计不足都是错误的。

务实而优秀的开发人员会考虑Web应用的上下文，不会增加任何不必要的复杂性。认真分析所有应用的非功能需求是避免构建错误的关键前提。

Compojure就是那种不妄想征服世界的Web框架。对于Web仪表板、操作监控，以及很多更加注重简单性和开发速度、而不是大规模扩展能力及其他非功能需求的简单任务来说，Compojure是非常理想的选择。从这种描述中你应该能猜出来，Compojure介于多语言编程金字塔的领域特定层和动态层之间。

在这一节我们会搭建一个简单的Hello World应用，然后讨论Compojure把Web应用串起来的简单规则。在用这些规则搭建示例程序之前，先介绍一个实用的Clojure HTML类库（Hiccup）。

如图13-6所示，Compojure构建在Ring框架之上，Ring框架是Clojure连接到Jetty Web容器的中间件。但使用Compojure/Ring并不需要对Jetty有多深入的了解。我们先用一个简单的Hello World作为Compojure的入门应用吧。

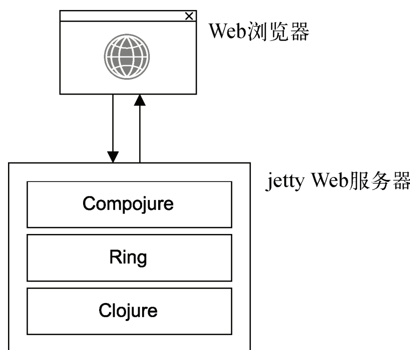


图13-6　Compojure和Ring

13.6.1　Hello Compojure

开始一个新的Compojure项目非常容易，因为Compojure跟Leiningen的工作流程自然融合。如果你还没装Leiningen，也没看第12章中的那一节，那你现在就应该去把这两件事做了，因为接下来的内容要求你熟悉Leiningen。

要开始一个新项目，只要执行一个普通的Leiningen命令：

```
lein new hello-compojure
```

在project.clj中可以轻松指明项目的依赖项。代码清单13-8显示了如何在project.clj文件中指定Hello World项目的依赖项。

代码清单13-8 简单的Compojure project.clj

```
(defproject hello-compojure "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.2.1"]
                 [compojure "0.6.2"]]
  :dev-dependencies [[lein-ring "0.4.0"]]
  :ring {:handler hello-compojure.core/app})
```

宏(defproject)跟第12章那个很像，不过多了两个元数据。

- :dev-dependencies确保开发人员可以在开发时使用lein命令。稍后我们讨论lein ring server时你就能见到实例了。
- :ring引入了Ring类库所需的挂钩。它将Ring特定的元数据映射为参数。

这个例子中给Ring传入了一个:handler属性。看起来它希望得到hello-compojure.core命名空间中的app符号。我们来看看代码清单13-9中core.clj中对它的声明，以便找出它们是如何相互配合的。

代码清单13-9 Compojure Hello World中简单的core.clj文件

```
(ns hello-compojure.core
  (:use compojure.core)
  (:require [compojure.route :as route]
            [compojure.handler :as handler]))

(load "hello")

(defroutes main-routes
  (GET "/" [] (page-hello-compojure))          <-- 主路由定义

  (route/resources "/")
  (route/not-found "Page not found"))

(def app (handler/site main-routes))            <-- 注册路由
```

这种把关联信息和其他信息保存在core.clj中的惯例非常实用。当有URL请求时再加载一个包含对应函数（*页面函数*）的单独文件很简单。这确实只是一个为了提高可读性，简单实现关注点分离的惯例。

Compojure使用了一组规则，称为*路由*，来确定如何处理接入的HTTP请求。这些规则是由Compojure依赖的Ring框架提供的，它们既简单又实用。你可能已经猜出来了，规则GET"/"告诉Web服务器如何处理对根URL的GET请求。我们下一节会对路由做更多的讨论。

为了完成这个例子的代码，还需要在src/hello_compojure目录中创建hello.clj文件。在这个文件中要定义一个如下所示的页面函数(page-hello-compojure)：

```
(ns hello-compojure.core)

(defn page-hello-compojure [] "<h1>Hello Compojure</h1>")
```

这个页面函数是个常规的Clojure函数，它会返回一个字符串作为HTML文档的<body>标签中的内容，而这个文档会作为响应的一部分返回给用户。

让我们把这个例子跑起来。在Compojure中这是个十分简单的操作。先确保所有依赖项都装上了：

```
ariel:hello-compojure boxcat$ lein deps
Downloading: org/clojure/clojure/1.2.1/clojure-1.2.1.pom from central
Downloading: org/clojure/clojure/1.2.1/clojure-1.2.1.jar from central
Copying 9 files to /Users/boxcat/projects/hello-compojure/lib
Copying 17 files to /Users/boxcat/projects/hello-compojure/lib/dev
```

到目前为止一切都好。现在需要把它跑起来，可以用Ring提供的`ring server`方法。

```
ariel:hello-compojure boxcat$ lein ring server
2011-04-11 18:02:48.596:INFO::Logging to STDERR via org.mortbay.log.StdErrLog
2011-04-11 18:02:48.615:INFO::jetty-6.1.26
2011-04-11 18:02:48.743:INFO::Started SocketConnector@0.0.0.0:3000
Started server on port 3000
```

这会启动一个简单的Ring/Jetty Web服务器（默认端口3000），以实现快速反馈。默认情况下，这个服务器会自动重载被修改的文件。

警告　需要知道开发服务器的重载是在文件这一层实现的。这意味着正在运行的服务器可能会因为重新加载页面导致其状态被冲掉（或更糟，被部分冲掉）。如果你怀疑发生了这种情况，并因此出现了问题，应该关掉服务器重新启动。启动Ring/Jetty很快，应该不会对开发时间有太大影响。

如果用浏览器访问开发机上的3000端口（或本机http://127.0.0.1:3000），应该会看到页面中显示出了“Hello Compojure”。

13.6.2　Ring和路由

我们来看看如何配置Compojure应用的路由。路由的定义应该能让你想到一种领域特定语言：

```
(GET "/" [] (page-hello-compojure))
```

这些路由规则应当被看做匹配接入请求的规则。其构成方式非常简单：

```
(<HTTP method> <URL> <params> <action>)
```

- HTTP方法，通常是`GET`或`POST`，但Compojure也支持`PUT`、`DELETE`和`HEAD`。如果要匹配这条规则，这个HTTP方法必须跟传入的请求相匹配。
- URL，请求对应的URL。如果要匹配这条规则，这个URL必须跟传入的请求相匹配。
- 参数，一个表示参数应该如何处理的表达式。很快我们就会对它展开讨论。
- 动作，与这条规则匹配时返回的表达式（通常表示为传入参数的函数调用）。

对这些规则的匹配按从上到下的顺序逐一比对，直到找到匹配项。Compojure会执行第一个匹配项的动作，表达式的值会作为返回文档`<body>`标签中的内容。

Compojure中规则的定义很灵活。比如说，创建一个从URL中提取函数参数的规则非常简单。我们来改一下代码清单13-5中的Hello World路由：

```
(defroutes main-routes
  (GET "/" [] (page-hello-compojure))
  (GET ["/hello/:fname", :fname #"[a-zA-Z]+" ]
➥ [fname] (page-hello-with-name fname))

  (route/resources "/")
  (route/not-found "Page not found"))
```

这个新规则只匹配包含`/hello/<名称>`的URL。其中的名称只能包含字母（大写、小写或大小写组合都行），这是由Clojure的正则表达式`#"[a-zA-Z]+"`限定的。

如果匹配了这一规则，Compojure会以匹配的名称为参数调用`(page-hello-with-name)`。函数定义非常简单：

```
(defn page-hello-with-name [fname]
  (str "<h1>Hello from Compojure " fname "</h1>"))
```

只有非常简单的应用才能用这种内联HTML，否则很快就会变成一种痛。好在有Hiccup模块，它为需要输出HTML的Web应用提供了很多实用的功能。马上我们就去看看。

13.6.3 Hiccup

要在hello-compojure应用中挂上Hiccup，需要做三件事：

- 在project.clj上加上依赖项，如`[hiccup "0.3.4"]`；
- 再次运行`lein deps`；
- 重启Web容器。

很好。现在我们来看看在Clojure内部怎么用Hiccup写出更好的HTML形式。

Hiccup提供的关键形式之一是`(html)`。用它可以非常直接地编写HTML。下面是用Hiccup重写的`(page-hello-with-name)`：

```
(defn page-hello-html-name [fname]
  (html [:h1 "Hello from Compojure " fname]
        [:div [:p "Paragraph text"]]))
```

现在这些嵌套格式的HTML标签看起来很像Clojure代码，所以把它放到代码里自然多了。`(html)`形式以一个或更多的（标签）向量为参数，并且标签的嵌套深度不受限制。

接下来，我们会向你介绍一个稍微大一点儿的应用，一个给水獭投票的网站。

13.7 我是不是一只水獭

互联网上似乎有两件事永远都不会让人厌烦：在线投票和可爱的动物图片。有个创业公司想把这两件事结合起来，让人们给水獭图片投票，然后靠广告回报赚钱，他们雇了你。勇敢面对吧，这毕竟还算不上是创业公司所尝试过的最傻的主意。

我们先想想这个水獭投票网站所需的基本页面和功能：

- 网站首页应该展示两张水獭供用户选择；
- 用户应该能给自己喜欢的那只水獭投票；
- 应该有个单独的页面允许用户上传水獭的新照片；

- 应该有个仪表板页面显示每张水獭图片的当前得票。

图13-7中展示了如何安排构成应用的页面和HTTP请求。

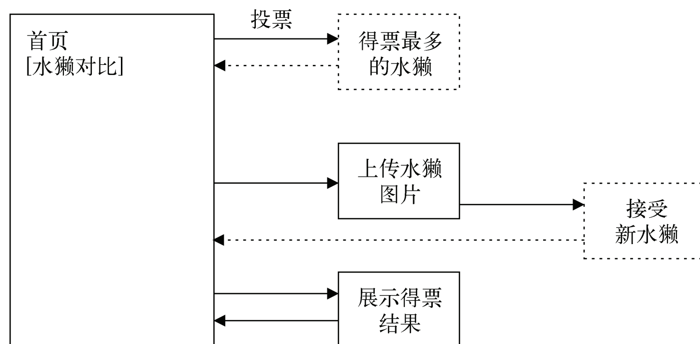


图13-7　“我是不是一只水獭？”的页面流

我们暂不考虑该应用的非功能性需求。

- 该网站不做访问控制。
- 对新上传的水獭图片文件不做安全检查。它们会以图片的形式在页面上显示，但上传对象的内容或安全性都没有经过检查。我们相信用户，他们不会上传任何不合适的东西。
- 该网站没有持久化。如果Web容器崩溃了，所有投票数据就都没了。但在应用启动时，它会扫描硬盘，预先填充水獭图片的存储。

尽管我们会在这一章中介绍其中的重要文件，但github.com上就有这个项目，你可能会发现那个更好用。

13.7.1　项目设置

要开始这个Compojure项目，需要定义基本项目：它的依赖项、路由，还有一些页面函数。我们先来看看project.clj文件，如代码清单13-10所示。

代码清单13-10　项目project.clj

```
(defproject am-i-an-otter "1.0.0-SNAPSHOT"
  :description "Am I an Otter or Not?"
  :dependencies [[org.clojure/clojure "1.2.0"]
                 [org.clojure/clojure-contrib "1.2.0"]
                 [compojure "0.6.2"]
                 [hiccup "0.3.4"]
                 [log4j "1.2.15" :exclusions  [javax.mail/mail
                                               javax.jms/jms
                                               com.sun.jdmk/jmxtools
                                               com.sun.jmx/jmxri]]
                 [org.slf4j/slf4j-api "1.5.6"]
                 [org.slf4j/slf4j-log4j12 "1.5.6"]]
  :dev-dependencies [[lein-ring "0.4.0"]]
  :ring {:handler am-i-an-otter.core/app})
```

这个文件中没什么新鲜玩意，除了log4j类库，其他在前面的例子里都有。

接下来我们看看core.clj文件里的连接和路由逻辑，如代码清单13-11所示。

代码清单13-11　core.clj的路由

```
(ns am-i-an-otter.core
  (:use compojure.core)
  (:require [compojure.route :as route]
            [compojure.handler :as handler]
            [ring.middleware.multipart-params :as mp]))

(load "imports")
(load "otters-db")
(load "otters")                                              导入函数

(defroutes main-routes                          ◁  主路由
  (GET "/" [] (page-compare-otters))
  (GET ["/upvote/:id", :id #"[0-9]+" ] [id] (page-upvote-otter id))
  (GET "/upload" [] (page-start-upload-otter))
  (GET "/votes" [] (page-otter-votes))

  (mp/wrap-multipart-params                     ◁  文件上传处理程序
   (POST "/add_otter" req (str (upload-otter req)
➥
 (page-start-upload-otter))))

  (route/resources "/")
  (route/not-found "Page not found"))

(def app
  (handler/site main-routes))
```

文件上传处理程序展示了一种新的参数处理方式。我们在下一小节还会展开来讲，但现在，可以把它看做“将整个HTTP请求传给页面函数处理”。

core.clj中的关联关系让你可以看清哪个页面函数跟哪个URL相关。所有页面函数都以page打头——这只是函数命名的惯例。

代码清单13-12给出了该应用程序的页面函数。

代码清单13-12　项目的页面函数

```
(ns am-i-an-otter.core
  (:use compojure.core)
  (:use hiccup.core))

(defn page-compare-otters []                    ◁  水獭比较页面
  (let [otter1 (random-otter), otter2 (random-otter)]
    (.info (get-logger) (str "Otter1 = " otter1 " ; Otter2 = "
➥ otter2 " ; " otter-pics))
    (html [:h1 "Otters say 'Hello Compojure!'"]
          [:p [:a {:href (str "/upvote/" otter1)}
                  [:img {:src (str "/img/"
➥ (get otter-pics otter1))} ]]]
          [:p [:a {:href (str "/upvote/" otter2)}
                  [:img {:src (str "/img/"
➥ (get otter-pics otter2))} ]]]
```

```
            [:p "Click " [:a {:href "/votes"} "here"]
                " to see the votes for each otter"]
            [:p "Click " [:a {:href "/upload"} "here"]
                " to upload a brand new otter"])))

(defn page-upvote-otter [id]                    <- 处理投票
  (let [my-id id]
    (upvote-otter id)
    (str (html [:h1 "Upvoted otter id=" my-id]) (page-compare-otters))))

(defn page-start-upload-otter []               <- 选择水獭上传
  (html [:h1 "Upload a new otter"]
        [:p [:form {:action "/add_otter" :method "POST"
➥ :enctype "multipart/form-data"}                <- 设置表单
            [:input {:name "file" :type "file" :size "20"}]
            [:input {:name "submit" :type "submit" :value "submit"}]]]
        [:p "Or click " [:a {:href "/"} "here" ] " to vote on some otters"]))

(defn page-otter-votes []                      <- 显示投票结果
  (let []
    (.debug (get-logger) (str "Otters: " @otter-votes-r))
    (html [:h1 "Otter Votes" ]
          [:div#votes.otter-votes
            (for [x (keys @otter-votes-r)]
              [:p [:img {:src (str "/img/" (get otter-pics x))} ]
➥ (get @otter-votes-r x)])])))
```

代码中还有两个Hiccup特性。第一个可以对一组元素进行循环，在这儿是刚上传的水獭图片。Hiccup在下面的代码片段中表现得非常像简单的模板语言（带有嵌入的(for)形态）：

```
[:div#votes.otter-votes
  (for [x (keys @otter-votes-r)]
    [:p [:img {:src (str "/img/" (get otter-pics x))} ]
➥ (get @otter-votes-r x)])]
```

第二个特性是:div#votes.otter-votes语法。这是指明某一标签的id和class属性的快捷办法。它会变成HTML标签<div class="otter-votes" id="votes">。开发人员可以借此把最可能由CSS使用的属性分离出来，不会让HTML结构变得太乱。

CSS和其他代码（比如JavaScript源文件）通常会放在静态内容目录中等待读取。在Compojure项目中默认是在resources/public目录下。

HTTP方法的选择

水獭投票这个例子在架构上有缺陷。我们为投票页面指定的路由规则是GET规则。这是错误的。

应用程序绝不应该用GET请求修改服务器端的状态（比如水獭的投票数）。因为Web浏览器在觉得服务器没有响应时是可以重发GET请求的（比如当请求进来时它正因为垃圾收集而暂停呢）。这一重发请求的行为可能会导致同一水獭收到重复投票，可实际上用户只点了一次。对于电子商务应用来说，这会引发灾难！

记住这条原则：有意义的服务器端状态绝不能用GET请求修改。

我们已经看过了关联起来的应用和它的路由，以及页面函数。我们再来看一些处理水獭投票的后台函数，继续讨论这个应用。

13.7.2 核心函数

在讨论应用的核心功能时，我们提到应用应该扫描图片目录找出磁盘里已有的水獭图片。代码清单13-13是扫描目录并进行预填充的代码。

代码清单13-13 目录扫描函数

```
(def otter-img-dir "resources/public/img/")
(def otter-img-dir-fq
  (str (.getAbsolutePath (File. ".")) "/" otter-img-dir))
(defn make-matcher [pattern]
  (.getPathMatcher (FileSystems/getDefault) (str "glob:" pattern)))

(defn file-find [file matcher]                                  ; 如果匹配，返回去掉两边空格的文件名
  (let [fname (.getName file (- (.getNameCount file) 1))]
    (if (and (not (nil? fname)) (.matches matcher fname))
      (.toString fname)                                         ; 用(toString)启用:img标签
      nil)))

(defn next-map-id [map-with-id]                                 ; 取下一个水獭的ID
  (+ 1 (nth (max (let [map-ids (keys map-with-id)]
    (if (nil? map-ids) [0] map-ids))) 0 )))

(defn alter-file-map [file-map fname]                           ; 修改函数并将文件名加到映射中
  (assoc file-map (next-map-id file-map) fname))

(defn make-scanner [pattern file-map-r]                         ; 返回扫描器
  (let [matcher (make-matcher pattern)]
    (proxy [SimpleFileVisitor] []
      (visitFile [file attribs]                                 ; 在所有文件上执行的回调函数
        (let [my-file file,
              my-attrs attribs,
              file-name (file-find my-file matcher)]
          (.debug (get-logger) (str "Return from file-find " file-name))
          (if (not (nil? file-name))
            (dosync (alter file-map-r alter-file-map file-name) file-map-r)
            nil)
          (.debug (get-logger)
➥ (str "After return from file-find " @file-map-r))
          FileVisitResult/CONTINUE))

      (visitFileFailed [file exc] (let [my-file file my-ex exc]
        (.info (get-logger)
          (str "Failed to access file " my-file " ; Exception: " my-ex))
        FileVisitResult/CONTINUE)))))

(defn scan-for-otters [file-map-r]
  (let [my-map-r file-map-r]
    (Files/walkFileTree (Paths/get otter-img-dir-fq
➥ (into-array String [])) (make-scanner "*.jpg" my-map-r))
    my-map-r))

(def otter-pics (deref (scan-for-otters (ref {}))))             ; 设置水獭图片
```

这段代码的入口是(scan-for-otters)。它用Java 7中的Files类从otter-img-dir-fq开始遍历文件系统，并返回一个映射。这里用了一个简单的惯例，以-r结束的标记名称表示这是对某个结构的引用。

遍历文件的代码是SimpleFileVisitor类（在java.nio.file包中）的Clojure代理，这个类在第2章就出现过。我们自行实现了其中两个方法：(visitFile)和(visitFileFailed)，对这个例子来说足够了。

其他有趣的函数是实现投票功能的那些，如代码清单13-14所示。

代码清单13-14　水獭投票函数

```
(def otter-votes-r (ref {}))

(defn otter-exists [id] (contains? (set (keys otter-pics)) id))

(defn alter-otter-upvote [vote-map id]
  (assoc vote-map id (+ 1 (let [cur-votes (get vote-map id)]
    (if (nil? cur-votes) 0 cur-votes)))))

(defn upvote-otter [id]
  (if (otter-exists id)
    (let [my-id id]
      (.info (get-logger) (str "Upvoted Otter " my-id))
      (dosync (alter otter-votes-r alter-otter-upvote my-id)
➥ otter-votes-r))
      (.info (get-logger) (str "Otter " id " Not Found " otter-pics))))

(defn random-otter [] (rand-nth (keys otter-pics)))

(defn upload-otter [req]
  (let [new-id (next-map-id otter-pics),
        new-name (str (java.util.UUID/randomUUID)
➥ ".jpg"),                                          ◁ 赋予随机文件名
        tmp-file (:tempfile
➥ (get (:multipart-params req) "file"))]            ◁ 提取临时文件
    (.debug (get-logger) (str (.toString req) " ; New name = "
➥ new-name " ; New id = " new-id))
    (ds/copy tmp-file (ds/file-str
➥ (str otter-img-dir new-name)))                     ◁ 复制到文件系统中
    (def otter-pics (assoc otter-pics new-id new-name))
    (html [:h1 "Otter Uploaded!"])))
```

在(upload-otter)函数中处理的是完整的HTTP请求映射。其中有很多信息可供Web开发人员使用，不过有些可能是你已经熟悉的了：

```
{:remote-addr "127.0.0.1",
 :scheme :http,
 :query-params {},
 :session {},
 :form-params {},
 :multipart-params {"submit" "submit", "file" {:filename "otter_kids.jpg",
     :size 122017, :content-type "image/jpeg", :tempfile #<File /var/tmp/
     upload_646a7df3_12f5f51ff33__8000_00000000.tmp>}},
 :request-method :post,
 :query-string nil,
```

```
:route-params {},
:content-type "multipart/form-data; boundary=----
    WebKitFormBoundaryvKKZehApamWrVFt0",
:cookies {},
:uri "/add_otter",
:server-name "127.0.0.1",
:params {:file {:filename "otter_kids.jpg", :size 122017, :content-type
    "image/jpeg", :tempfile #<File /var/tmp/
    upload_646a7df3_12f5f51ff33__8000_00000000.tmp>}, :submit "submit"},
:headers {"user-agent" "Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_6;
    en-US) AppleWebKit/534.16 (KHTML, like Gecko) Chrome/10.0.648.205
   Safari/534.16", "origin" "http://127.0.0.1:3000", "accept-charset" "ISO-
   8859-1,utf-8;q=0.7,*;q=0.3", "accept" "application/xml,application/
   xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5", "host"
   "127.0.0.1:3000", "referer" "http://127.0.0.1:3000/upload", "content-
   type" "multipart/form-data; boundary=----
   WebKitFormBoundaryvKKZehApamWrVFt0", "cache-control" "max-age=0",
   "accept-encoding" "gzip,deflate,sdch", "content-length" "122304",
   "accept-language" "en-US,en;q=0.8", "connection" "keep-alive"},
:content-length 122304,
:server-port 3000,
:character-encoding nil,
:body #<Input org.mortbay.jetty.HttpParser$Input@206bc833>}
```

从这个请求映射中能看到容器已经把上传的文件内容放到了`/var/tmp`的临时文件中。可以通过`(:tempfile (get (:multipart-params req) "file"))`访问相应的`File`对象。然后简单地用`clojure.contrib.duck-streams`中的`(copy)`函数把它保存到文件系统中。

水獭投票不大，但它是一个完整的应用程序。在本节开头提出的功能性和非功能性需求的限定下，它的表现符合我们的预期。我们对Compojure及一些相关类库的探索就到此为止了。

13.8 小结

快速Web开发应该是所有优秀Java开发人员都能做的事情。但如果选了糟糕的语言或框架，很快你就会落在Rails和PHP这种非Java/JVM技术人员的后面。尤其是静态类型的编译型Java语言，它有时候不是做Web开发的理想选择。相反，选对了语言或框架，就可以在保证质量的前提下快速实现新功能，助你攀上Web开发食物链的顶端，可以针对用户所需快速做出反应。

优秀的Java开发人员不希望扔掉强大灵活的JVM。幸好，随着JVM上的其他语言及其Web框架的出现，你可以留着它了！像Grails和Compojure这样的动态层框架提供了你所需要的快速Web开发能力。

特别是Grails，可以非常迅速地搭建一个完整的（UI到数据库）原型，然后开发人员就可以用强大的展示层技术（GSP）、存储层技术（GORM）和一大堆实用的插件把各个部分撑起来。

Compojure可以很自然地跟Clojure编写的项目相结合。也非常适合用来向Java或其他语言的项目中添加小型Web组件，比如仪表板和操作控制台。简洁的代码和快速的开发能力是Compojure的主要优势。

我们就这样学习了JVM多语言编程的各种示例，走到了各章的结尾。在最后一章，我们会把所有的线索都抓到一起，看一些超前的知识。那里有超出我们现有经验之外的挑战，但现在我们掌握的工具已经可以处理它们了。

第 14 章 保持优秀

14

本章内容

- Java 8对开发者的意义
- 多语言编程的未来
- 并发性的发展分方向
- JVM层的新特性

要走在时代的前列，优秀的Java开发人员总是应该对即将到来的东西保持清醒的认识。本书最后一章会讨论几个在我们看来指引Java语言及平台未来发展方向的主题。

因为我们既没有TARDIS[①]也没有水晶球，所以本章的内容主要集中在据我们所知已经在开发的语言特性和平台修改上。也就是说这只能是当下的观点，客气的说法是这在某种程度上来说算是科幻作品。

撰写本书过程中我们所讨论的观点只是代表将来的一种可能。事情如何发展还有待时间验证。毫无疑问的是，在某些重要方式上事情的发展会跟我们此处的讨论有所不同，并且以非常有趣的方式到达那一点。通常都是这样。

让我们先去看看第一个主题吧，快速浏览一下很可能出现在Java 8中的一些主要特性。

14.1 对 Java 8 的期待

2010年秋，Java SE执行委员会商议决定执行B计划。这个决定是尽快发布Java 7，并把一些主要特性延迟到Java 8中。这一结论是在对社区进行广泛征询和投票后得出的。

在Java 7中发起的某些特性已经被推到Java 8中了，还有些特性已经缩减了范围以便为将来的特性打下基础。在这一节中，我们会对一些期望Java 8突出的特性做简要介绍，包括那些被延迟的特性。在这一阶段，没有什么是板上钉钉的，特别是语法。所有示例代码都只是初步构想，可能跟Java 8的最终写法差别很大。欢迎来到风口浪尖！

① TARDIS是英国科幻电视剧《神秘博士》（*Doctor Who*）中的时间机器和宇宙飞船，是时间和空间相对维度（Time and Relative Dimension In Space）的缩写。——译者注

14.1.1　lambda 表达式（闭包）

将在Java 8中构建的Java 7特性中最有代表性的是MethodHandles和invokedynamic。它们本身是非常实用的特性（在Java 7中，invokedynamic主要用在语言和框架实现上）。

在Java 8中，这些特性是将lambda表达式引入Java语言的基础。可以这样理解，lambda表达式跟前面在备选语言中讲的函数字面值类似，它们也能用来解决我们在前面重点强调的那类问题。

就Java 8的语法而言，还需要决定lambda表达式在代码中如何表示。但其基本特性已经确定下来了，所以我们先来看看基本的Java 8语法，如代码清单14-1所示。

代码清单14-1　在Java中用lambda表达式实现Schwartzian变换

```
public List<T> schwarz(List<T> x, Mapper<T, V> f) {
  return x.map(w -> new Pair<T,V>(w, f.map(w)))
          .sorted((l,r) -> l.hashed.compareTo(r.hashed))
          .map(l -> l.orig).into(new ArrayList<T>());
}
```

schwartz()方法看起来应该眼熟，它是在10.3节中用闭包实现的Schwartzian变换。代码清单14-1展示了Java 8中lambda表达式的下列基本语法：

- 在lambda表达式前部有个参数列表；
- 组成lambda表达式的主体代码块用括号括起来；
- 用箭头（->）来分隔参数列表和lambda表达式的主体；
- 参数列表中参数的类型是可推断的。

第9章中Scala的函数字面值和这个写法很像，所以这种语法应该不会让你觉得特别陌生。代码清单14-1中的lambda表达式非常短，全都只有一行。实际上，lambda表达式是可以包含多行代码的，其主体甚至可以很大。经过初步分析，那些适于改造成lambda表达式的代码改造后的长度都应该在1到5行之间。

代码清单14-1中还介绍了另外一个新特性。变量x的类型是List<T>。我们在x上调用了方法map()。map()方法接受了一个lambda表达式作为其参数。停！List接口根本就没有map()方法，并且在Java 7及之前都不存在lambda表达式。

我们来仔细看看这个问题是如何解决的。

1. 扩展和默认方法

我们所面临的问题本质是：怎么向已有接口中添加方法以使其“lambda化”而又不破坏其向后兼容性？

答案来自于Java的一个新特性：扩展方法。它可以为没有提供扩展方法的接口实现提供一个可用的默认方法。

这些默认的方法实现必须在接口本身内部定义。比如跟List搭配的AbstractList，跟Map搭配的AbstractMap，跟Queue搭配的AbstractQueue。这些类是为各自的接口存放新的扩展方法默认实现的理想之所。Java内置的集合类是扩展方法和lambda化的主要应用场景，但看起来这种模型在最终用户代码中也适用。

Java怎么实现扩展方法

扩展方法会在类加载时进行处理。当加载一个带有扩展方法的接口实现时，类加载器会检查它是否实现了自己的扩展方法。如果没有，类加载器会定位到默认方法，并把一个桥接方法插入新加载类的字节码中。这个桥接方法会用`invokedynamic`调用默认方法。

扩展方法无需打破向后兼容性就可以让发布后的接口得到进化。开发人员可以借此带着lambda表达式为老旧的API注入新的活力。但lambda表达式对于JVM来说是什么呢？是对象吗？如果是，它们的类型是什么？

2. SAM转换

lambda表达式提供了一种紧凑的办法，可以声明少量内联代码并将其作为数据传递。也就是说lambda是一个对象，就像我们在本书第三部分中对lambda表达式和函数字面值的解释一样。具体说来，你可以把lambda表达式当做`Object`的一个子类，它没有参数（因此也没有状态），只有一个方法。

还有一种理解这个问题的方式：通过术语SAM（Single Abstract Method，单例抽象方法）。SAM的概念在各种Java API中都有体现，是一种常见主题。很多API中都有只声明了一个单例方法的接口。`Runnable`、`Comparable`、`Callable`和`ActionListener`之类的监听器都只声明名了一个方法，因此都算SAM类。

在刚开始用lambda表达式时，可以把它们当做语法糖——为给定接口编写匿名实现的简便写法。过一段时间后，你可以掌握更多的函数式技术，甚至可能会从Scala或Clojure中把自己喜欢的技巧引入Java代码中。学习函数式编程是个循序渐进的过程：从集合的映射、排序和过滤技术开始学起，然后慢慢向外开疆拓土。

现在让我们进入下一个大主题：模块化编程，它正在Jigsaw项目的支持下如火如荼地展开。

14.1.2　模块化（拼图 Jigsaw）

处理classpath有时毫无疑问是不太理想的。围绕着JAR文件和classpath构建的生态系统有些众所周知的问题：

- JRE自身的规模就比较大；
- JAR文件提倡整体式部署模型；
- 有些繁琐且极少会用到的类仍然必须加载；
- 启动慢；
- classpath是脆弱的野兽，并且跟机器上的文件系统结合得过于紧密；
- classpath基本上是一个扁平化的命名空间；
- JAR不具有固有的版本；
- 即便逻辑上没有关联的类之间也有复杂的相互依赖关系。

要解决这些问题，需要一个新的模块系统。但要先解决架构上的问题。其中最重要的如图14-1所示。

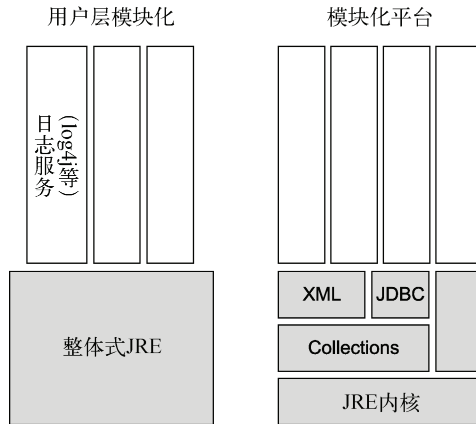


图14-1　模块化系统的架构选择

我们是应该引导VM然后再使用用户层模块化系统（比如OSGi），还是应该彻底迁移到模块化平台上？

后一种方式需要启动一个能支持模块的最基本的“内核”VM，然后根据启动应用程序的需要添加特定的模块。这要求对VM，以及JRE中很多现有的类做颠覆性的修改，但潜在收益更大。

- JVM应用程序的启动时间可以跟shell和脚本语言相媲美。
- 能显著降低应用程序部署的复杂性。
- 针对性的Java安装所占用的资源可以显著减少（对硬盘、内存和安全性都有积极影响）。如果你不需要CORBA或RMI，就不用装！
- 可以以更加灵活的方式升级Java安装。如果在Collections中发现了一个严重的bug，只有那个模块需要升级。

撰写本书时看起来Jigsaw项目会选择第二种方式。但在它发布并能投入使用之前，还有很长的路要走。下面是一些仍在讨论的重要问题：

- 平台或应用的正确发布单元是什么？
- 是不是需要一种跟包和JAR都不同的新结构？

这一设计决策的影响极为重要：Java无处不在，因而这种模块化的设计要渗透到所有地方。它也要支持跨OS平台。

Java平台最起码要能在Linux、Solaris、Windows、Mac OS X、BSD Unix和AIX上部署模块化应用。这些平台中有些有需要Java模块集成的包管理器（比如Debian的apt、Red Hat的rpm，以及Solaris包）。而其他平台，比如Windows，没有可供Java使用的包管理系统。

这一设计还有其他的限制。不过这一领域已经有一些成熟的项目了：比如Maven和Ivy这样的

依赖项管理系统，还有发起倡议的OSGi。新的模块化系统应该尽一切可能跟现有项目集成，即便在完全的集成和兼容被证明不可能之后，也应该提供一个顺畅的升级途径。

不管将来会怎样，Java 8的发布应该会给Java应用程序的交付和部署带来革命性的变化。

我们去看看JDK 8应该给JVM的其他公民带来的一些特性，包括我们在前面研究的那些语言。

14.2 多语言编程

从第5章开始，你已经无数次见证了JVM作为语言运行时平台的奇妙。第1章介绍的OpenJDK项目在Java 7的发布周期中成了Java的参考实现。非常有趣的是JVM已经发展成了一个语言无关的、真正支持多语言编程的虚拟机。

特别是随着Java 7的发布，Java语言丧失了在VM上的特权。平台上的所有语言现在都一视同仁。因此人们对添加之于备选语言非常重要、而对Java本身只有边际效益的VM特性表现出了强烈的兴趣。

这一工作是在达芬奇机（Da Vinci Machine）子项目中开展的，这一项目也叫做mlvm（多语言VM）。在这一项目中培育出的特性会被引入源码主干中。5.5节中的`invokedynamic`就是这样的例子，但还有很多对非Java语言非常实用的其他特性。也有需要解决的问题。

我们先来看看这些语言特性中的第一个：不同的语言运行在同一个JVM中彼此进行无障碍交流的办法。

14.2.1 语言的互操作性及元对象协议

语言平等是向了不起的多语言编程环境迈出的重要一步，但一些棘手的问题仍然存在。其中主要是不同的语言有不同的类型系统。Ruby的字符串是可修改的，而Java的不可修改。Scala把所有东西都当成对象，即便是在Java里作为原始类型的实体也是如此。

处理这些差异，并为同一JVM内的不同语言提供更好的交互和互操作方式，是目前尚未解决的问题，也是正在积极处理有望近期解决的问题。

想象一个将来要做的Web应用：其核心部分可能是Java代码，Web部分是用Compojure写的（即Clojure），所用的JSON处理类库是用纯粹的JavaScript写的，而你想用ScalaTest中一些很酷的TDD功能来对它进行测试。

这形成了一个JavaScript、Clojure、Scala和Java彼此之间都会直接调用的局面。对JVM语言能够互操作并以一种标准的方式调用彼此对象的需求会随着时间逐渐增强。社区内的广泛共识是需要一种元对象协议（Metaobject Protocol，MOP），以便所有这些语言都能以一种标准的方式工作。MOP可以看做是一种在代码内描述特定的语言如何实现面向对象及相关问题的办法。

要实现这一目标，我们需要想想那些能让某种语言中的对象能在另外一种语言中使用的办法。一种简单的方式是把它转换成其他语言中的本地类型（或甚至在外部运行时中创建一个新的“影子”对象）。这种办法简单，但有严重的问题：

- 所有语言必须都有一个通用的“主”接口（或超类），语言内的所有类型都必须实现它（比如JRuby中的`IRubyObject`）；
- 如果用影子对象，那会增加很多内存分配，性能也会受影响。

相反，我们可以考虑为外部运行时构建一个服务作为入口。这一服务会提供一个接口，某一运行时可以通过该接口对外部运行时中的对象执行标准操作，比如：

- 在其他语言运行时中创建一个新对象并返回对它的引用；
- 访问（获取方法或设置方法）外部对象的属性；
- 调用外部对象上的方法，并返回结果；
- 将外部对象转换成不同的相关类型；
- 访问外部对象的其他能力，对一些语言来说可能和方法调用的语义有所不同。

在这样的系统中，可以通过在外部运行时上调用`navigator`来访问外部方法或属性。调用者需要提供一种办法来标识要访问的方法：`someMethod`。通常是个字符串，但某些情况下也可能是`MethodHandle`。

```
navigator.callMethod(someObject, someMethod, param1, param2, ...);
```

要让这种办法起作用，所有协作语言运行时中的navigator接口必须都一样。实际上，语言之间的真实联系很可能是用`invokedynamic`建立起来的。

接下来我们去看看多语言JVM和Java 8的模块化子系统组合起来是个什么样子。

14.2.2　多语言模块化

随着Jigsaw和平台模块化的出现，不仅仅是Java才会从模块化中受益（并需要参与进来）。其他语言也能加入其中有所表现。

可以想象，navigator接口及其辅助类很可能会成为一个模块，对某一非Java语言运行时的支持将会由一个或多个模块实现。图14-2中展示了这一模块系统看起来是什么样子。

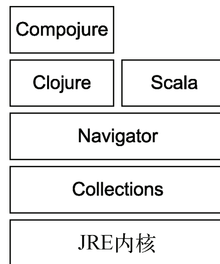


图14-2　实现了多语言解决方案的模块

如你所见，我们可以用模块系统搭建包含多种语言的应用程序。Clojure模块提供基本的Clojure平台，Compojure模块引入运行webapp所需的组件，包括特定版本的JAR，在别处运行时

这些JAR可能会用不同的版本。Scala及其XML也出现了，为了实现Scala和Clojure之间的互操作，Navigator模块也出现了。

在下一节中，我们会讨论非Java语言在平台上的爆炸性涌现所推动的另一个编程趋势：并发。

14.3 未来的并发趋势

20世纪的语言不一定能充分发挥21世纪的硬件的作用。我们之前讨论的内容已经多次暗示了这一点。在第6章讨论晶体管数量增长的摩尔定律时（6.3.1节），由于一个非常重要的原因，我们当时仅简要讨论了一下。那就是摩尔定律、性能和并发之间的相互作用，这也是我们的第一个主题。

14.3.1 多核的世界

尽管晶体管数量的爆炸性增长跟预测一样，但内存的访问速度却没能跟上。在20世纪90年代和21世纪头几年，芯片设计者用大量晶体管解决相对较慢的主存问题。

就像第6章讨论的，这可以确保有稳定的数据流供核心处理。但这根本是一场输掉的战斗：在晶体管上提升的速度变得越来越边际化。这是因为过去用的技术（比如指令级并行和投机式执行）现在已经把容易提升的速度榨光了，投机性变得越来越强。

最近这些年，业界已经把注意力转到了用晶体管在每个芯片上提供更多处理器内核。现在几乎所有笔记本或台式机都至少是双核的，4核和8核也很常见。在更高端的服务器上，可以找到6核或8核的芯片，整机能达到32（或更多）个核心。多核世界就在这里，要充分发挥它的作用，需要以串行处理更少的风格编程。那需要得到语言和运行时环境的支持。

14.3.2 运行时管理的并发

我们已经看到了未来并发编程的开始。在Scala和Clojure中，我们讨论了与Java的线程和锁模型有很大差异的并发观点：Scala的actor模型和Clojure的软件事务型内存方式。

Scala的actor模型允许在运行的代码块之间发送消息，而这些代码可能运行在完全不同的核心上（甚至有允许actor远程运行的扩展）。这就是说代码完全是按以actor为中心的方式编写的，因此在多核机器上扩展非常简单。

跟Scala actor一样，Clojure中的代理填补了相同的生态位[①]，但Clojure中还有只能在一个内存事务中修改的共享数据（refs）——软件事务型内存机制。

在这两种并发中，都能见到一种新概念的萌芽：由运行时（而不是开发人员）管理并发。尽管JVM提供了线程调度的底层服务，但它没有提供管理并发程序的高层结构。

这个缺陷在Java语言中能看出来，导致Java程序员基本上在用JVM的底层模型。

① 生态位（Ecological niche），又称小生境、生态区位、生态栖位或是生态龛位，是一个物种所处的环境以及其本身生活习性的总称。每个物种都有自己独特的生态位，区别于其他物种。——译者注

别对Java并发太过苛求

Java在1996年发布，它是从一开始就考虑并发的主流语言之一。经过业界15年的广泛实操，我们才对可变数据、默认共享状态和用协作锁强制排他执行这种模型的问题有所察觉。但发布Java 1.0的工程师没有这种福利。从很多方面来说，Java在并发上的首次尝试都是我们取得今天这种成就的基础。

现在有大把的代码撒在外面，再为Java做一种全新的机制来强制推行，还要跟现有代码无缝交互，这非常困难。所以大部分注意力都放在了为非Java的JVM语言找寻新的并发出路上。这些语言有两个重要特性：

- 以JMM为底层模型；
- 跟Java相比有“全新设计”的语言运行时，可以提供不同的抽象层（并且强制性更强）。

在VM层面出现更多的并发支持也不是不可能（下一节会讨论到），但目前来看主流还是在以JMM为基础的新语言上做创新，而不是修改底层的基础线程模型。

在JDK 8及以后的版本中，肯定能见到JVM的某些区域会发生变化。其中的一些变化顺延了Java 7的`invokedynamic`，这也是我们要讨论的下一主题。

14.4 JVM 的新方向

我们在第1章介绍了VMSpec（JVM规范）。这一文档确切指明了作为JVM标准实现的VM必须遵守的行为准则。当引入新行为时（比如Java 7的`invokedynamic`），所有实现都必须升级以支持新功能。

在这一节里，我们会谈到那些已经在讨论并有原型的各种修改最终实现的可能性。这项工作是在OpenJDK项目中开展的，该项目是Java参考实现的基础，也是Oracle JDK的起点。除了对规范的可能修改，我们也会涉及对OpenJDK/Oracle JDK代码的显著改动。

14.4.1 VM 的合并

在Oracle收购了Sun公司之后，它就拥有了两款非常强的Java虚拟机：HotSpot VM（Sun带的）和JRockit（之前收购的BEA带的）。

Oracle很快就决定不再同时维护两个VM来浪费资源，要把它们合并起来。HotSpot VM被选作基础，JRockit特性会在将来发布Java时谨慎地引进。

名称有什么关系？

这个合并后的VM没有官方名称，尽管VM粉和Java社区大部分都支持HotRockit这个名称。它也确实挺吸引人，但还是要看Oracle的营销部门同不同意。

所以这对咱开发人员来说，这有什么关系呢？你现在用的VM（很可能是HotSpot VM）将来会增加很多新特性，包括（但不限于）下面这些：

- 去掉PermGen，能防止一大类跟类加载有关的崩溃；
- 加强JMX代理的支持，能让你对运行的VM有更多深入的了解；
- 新的JIT编译方式，从JRockit中引入新的优化；
- 任务控制，提供有助于对生产型应用进行调优和分析的先进工具。这些工具中有些可能是需要付费的外加JVM组件，不包含在免费下载的发布包中。

去掉PermGen

就像6.5.2节说的，类的元数据当前保存在VM中一个的特殊内存区里（PermGen）。它很快就会被填满，特别是对于那些在运行时会创建大量类的非Java语言和框架而言。PermGen区不回收，耗光之后还会导致VM崩溃。有关人员正在开展工作，要把元数据保存在自有内存区中，让噩梦一般的“java.lang.OutOfMemory-Error: PermGen space”消息永远地成为过去。

还有很多的小改进全都是为了让VM更小、更快、更灵活。假定HotSpot上大约已经投入了1000人年的工作量，跟投入工作量更多的JRockit结合起来形成的VM前景一定更加光明。

除了合并VM，还有大量的新特性正在制作中。其中之一就是可能会增加称为协同程序的并发特性。

14.4.2 协同程序

Java和JVM语言程序员了解最多的并发形式就是多线程。它依靠JVM的线程调度服务在处理器核心上启动和停止线程，但线程没办法控制这个调度。出于这一原因，多线程被称为“抢占式多任务”，因为调度器可以抢占正在运行的线程，迫使它放弃对CPU的控制。

协同程序的基本思想是允许执行单元部分参与控制对它们的调度。具体来说，协同程序会像普通线程那样运行，直到它遇到了一个“退位”指令。这会导致协同程序把自己挂起，并允许另一个协同程序继续在它的地盘运行。当原来的协同程序再次得到机会运行时，它会从退位之后的下一条语句继续向下执行，而不会从方法开始的地方。

因为这种多线程的方式靠正在运行的协同程序的相互协作，间或将运行机会退让给其他协同程序，这种多线程处理被称为“协作式多任务”。

关于协同程序如何工作的确切设计仍然处于热烈讨论的阶段，没有哪个是肯定要被采纳的。一个可能的模型是在一个单例共享线程（或类似于`java.util.concurrent`里的线程池）中创建和调度协同程序，如图14-3所示。

正在执行协同程序的线程可能会被系统内的其他任何线程抢占，但JVM线程调度器不能强迫协同程序退位。也就是说，以相信执行池中所有其他协同程序为代价，协同程序就可以控制什么时候切换上下文。

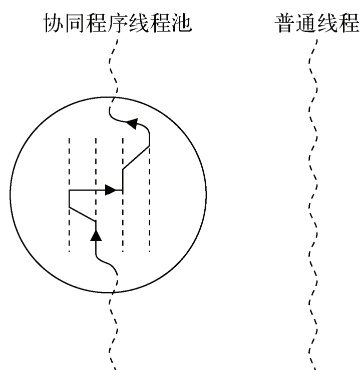


图14-3　一种可能的协同程序模型

这种控制意味着协同程序之间的同步可以做得更好。多线程代码必须构建复杂的锁策略来保护数据，但它很脆弱，因为上下文切换可能随时都会发生。这是我们在4.1节讨论的并发类型安全问题。相较而言，协同程序只要确保退位点数据的一致性，因为它知道其他任何时候自己都不会被抢占。

这个折中的额外担保是以相信其他线程为交换条件的，这是对某些线程编程问题的有益补充。一些非Java语言已经开始支持协同程序（或与之很贴近的概念“纤维”)，特别是Ruby和较新版的JavaScript。在VM层面增加协同程序的支持（但不一定是对Java语言）会对可以使用协同程序的语言有很大帮助。

在可能会实现的VM修改中，最后要讨论的是“元组”，这个VM特性提案对性能敏感的计算空间可能会产生很大的影响。

14.4.3　元组

在当今的JVM里，所有数据项不是原始类型就是引用类型（可能是引用对象或数组）。比较复杂的类型只能在类里定义，并传递对这些新类型实例对象的引用。这是一个简单而又相当优雅的模型，过去一直为Java服务得很好。

但要构建高性能系统，这个模型就会暴露几个缺陷。尤其是在游戏和金融软件这样的应用中，遇到这个简单模型局限性的情况十分常见。可以解决这个问题的办法之一就是采用元组。

元组（tuple）有时称为值对象，是能在原始类型和类之间架起桥梁的语言结构。像类一样，用元组可以定义包含原始类型、引用类型和其他元组的自定义复杂类型。像原始类型一样，在将它们传递给方法（或从方法中传递出来），保存在数组和其他对象中时，用的是整个值。如果你熟悉C（或.NET）环境，可以把它们看做结构（struct）的等价物。

我们来看一个例子：一个现有的Java API。

```
public class MyInputStream {
  public void write(byte[], int off, int len);
}
```

这让用户可以将特定数量的数据写到数组中的特定位置，很实用。但它设计得并不好。在理想的面向对象世界，偏移和长度应该被封装在数组内，并且无论是用户还是方法的实现者都应该不用再单独跟踪额外的信息。

实际上，在引入NIO时ByteBuffer就封装了这些信息。可惜这不是白来的，从ByteBuffer中创建新切片需要分配一个新对象，这会给垃圾收集子系统造成压力。尽管大多数垃圾收集器都非常擅长收集短命的对象，但在吞吐率非常高的延迟敏感环境中，这种分配操作会累加并最终导致应用出现令人无法接受的暂停。

如果我们能定义一个保存数组引用、偏移和长度的值对象（也就是元组）类型Slice会发生什么呢？在代码清单14-2中，我们会用新的tuple关键字来表示这个新概念。

代码清单14-2　作为元组的数组切片

```
public tuple Slice {
  private int offset;
  private int length;
  private byte[] array;

  public byte get(int i) {
    return array[offset + i];
  }
}
```

这个切片的构造结合了原始类型和引用类型的很多优势：

- Slice值可以复制到方法中，也可以从方法中复制出来，就跟手工传递数组的引用和int值一样有效；
- Slice元组在退出方法后会被清理掉（因为它们跟值类型一样）；
- 对偏移和长度的处理会干净地封装在元组中。

在日常编程中有很多类型会从元组的使用中受益，比如带有分子和分母的有理数、带有实部和虚部的复数，或者由ID和领域标识引用的用户主档（献给那些MMORPG迷们）。

在处理数组时元组也能对性能有所提升。现在的数组中要放同质的数据值集合——要么是原始类型，要么是引用类型。在使用数组时，元组允许我们对内存的布局做更多的控制。

来看一个例子：一个以原始类型long为键的简单散列表。

```
public class MyHashTable {
  private Entry[] entries;
}

public class Entry {
  private long key;
  private Object value;
}
```

在当前的JVM化身中，entries数组中只能放Entry实例的引用。调用者每次查找表中的key，在用传入的值与相关Entry实例的key比较之前，必须把Entry实例解引用。

当用元组实现时，则可以在数组内展开Entry类型，因此能够省掉访问key产生的解引用开销。图14-4展示了当前情况，以及使用元组后得到的改善。

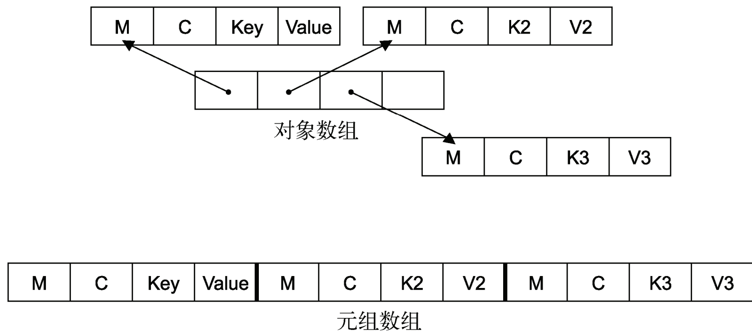


图14-4　JVM数组与元组

在考察元组的数组时，采用元组得到性能优势的关键之处也变得更加清晰。我们在第6章讨论过，大多数应用程序代码的性能都是由一级缓存的命中率决定的。在图14-4中，如果使用元组，扫描散列表的代码效率会更高。它不用再承担额外的缓存读取就能得到`key`值。这就是元组取得性能优势的本质——程序员在展开内存的数据时可以得到更优的空间局部性。[1]

对可能出现在Java和JDK 8中的新特性，我们的讨论就到此为止了。其中有多少能变成现实也只能等到快要发布时才知道。如果你对特性的演进感兴趣，可以加入OpenJDK项目和Java Community Process，参加这些特性的开发活动。如果你对它们还不熟悉，请找到这些项目并看看如何加入。

14.5　小结

Java 8会紧随Java 7的脚步。它会满载着各种改进而来，让开发人员可以更加高效地为现代化硬件编写代码，无论是最小的嵌入式设备还是最大的主机。

一种语言解决所有编程问题的神话已经破灭了。为了搭建有效的解决方案，比如高并发的交易系统，开发人员需要学习能跟核心Java代码互操作的新语言。

随着多核硬件和OS持续提供高度并行的编码架构，并发仍然是热门话题。想要解决海量数据、复杂运算或高速应用，这是一个必须跟进的领域。

Java VM被看做是当前最好的虚拟机。因为优秀的Java开发人员很可能会开辟高性能计算等新领域，所以有必要密切关注未来的发展态势。

如你所见，这里正在发生着巨变！我们认为Java生态系统正在经历一次规模宏大的涅槃，再过几年，优秀的Java开发人员必将光彩熠熠。

① 空间局部性（spatial locality），如果程序访问某个存储器地址后，又在较短时间内访问临近的存储器地址，则程序具有良好的空间局部性。两次访问的地址越接近，空间局部性越好。——译者注

附录 A java7developer：源码安装

在读一本新的技术书时，我们都喜欢真刀真枪地用代码做练习。它能帮我们正确理解书中的内容，光靠读代码达不到那种效果。

本书源码可在www.manning.com/evans/或www.java7developer.com/[1]处下载。我们会把你放代码的位置称为$BOOK_CODE。

本书中所有源码都放在java7developer项目中。它混合了Java、Groovy、Scala和Clojure的源码，以及它们的支持类库和资源。它不是那种典型的Java项目，你需要按照这个附录中的指令来构建它（即编译源码和运行测试）。我们会用Maven 3来执行各种构建周期目标，比如`compile`和`test`。

先来看看java7developer项目的源码布局。

A.1 java7developer 的源码结构

java7developer项目的结构遵守我们在第12章介绍的Maven规范，因此布局方式如下所示：

```
java7developer
|-- lib
|-- pom.xml
|-- sample_posix_build.properties
|-- sample_windows_build.properties
`-- src
    |-- main
    |   '-- groovy
    |       `-- com
    |           `-- java7developer
    |               `-- chapter8
    |   `-- java
    |       `-- com
    |           `-- java7developer
    |               `-- chapter1
    |               `-- ...
    |               `-- ...
    |   `-- resources
    |   `-- scala
    |       `-- com
```

① 也可在图灵社区（www.ituring.com.cn）本书网页免费注册下载。——编者注

```
    |           `-- java7developer
    |               `-- chapter9
    `-- test
        `-- java
            `-- com
                `-- java7developer
                    `-- chapter1
                    `-- ...
                    `-- ...
        `-- scala
            `-- com
                `-- java7developer
                    `-- chapter9
`-- target
```

按它的规范，Maven把主代码和测试代码分开了。它还为其他需要包含在构建中的文件设了个特殊的resources目录（比如日志记录的log4j.xml、Hibernate配置文件以及其他类似资源）。Maven的构建脚本是pom.xml文件，附录E中有对它的详细讨论。

Scala和Groovy源码跟Java源码的目录结构一样，只是Java的根目录是java，而它们的根目录分别是scala和groovy。Java、Scala和Groovy在Maven项目中可以排排坐，和睦相处，Clojure的源码处理起来稍有不同。Clojure大多数都是通过一个交互式环境处理的（所用的构建工具也不同，叫Leiningen），所以我们只是提供了一个clojure目录，用来存放Clojure源码，做练习的时候可以复制到Clojure REPL中。

在Maven构建运行之前不会创建target目录。构建产生的所有类、工件、报告和其他文件都会出现在这个目录下。

lib目录中放了些类库文件，以防Maven不能访问互联网下载所需类库。

看看项目结构，让自己熟悉一下各章的源码都放在哪里。一旦搞清楚源码的位置，就可以安装和配置Maven 3了。

A.2 下载并安装 Maven

可以到http://maven.apache.org/download.html下载Maven。在第12章的例子中，我们用的是Maven 3.0.3。如果你用的是*nix操作系统，请下载apache-maven-3.0.3-bin.tar.gz，如果是Windows，则下载apache-maven-3.0.3-bin.zip。文件下载完成后，只要选好目录把文件解压（untar/gunzip或unzip）就行了。

警告 跟很多Java/JVM相关软件的安装一样，在安装Maven的目录名称中也不要有空格，否则可能会出现`PATH`和`CLASSPATH`错误。比如说，如果你用的是Windows操作系统，不要把Maven装在C:\Program Files\Maven\这样的目录中。

在下载和解压完成后，接下来就是设置`M2_HOME`环境变量。在*nix系统中，需要加一些下面这样的东西：

```
M2_HOME=/opt/apache-maven-3.0.3
```

在Windows系统中是这样的：

```
M2_HOME=C:\apache-maven-3.0.3
```

你可能在想："为什么是M2_HOME而不是M3_HOME？毕竟这是Maven 3，对不对？"这是因为Maven的开发团队真的很想跟得到广泛应用的Maven 2保持兼容。

Maven需要Java JDK才能运行。1.5之后的版本都行（当然，到这一阶段，你已经装好JDK 1.7了）。还需要确保环境变量JAVA_HOME已经设置好了——如果已经装好Java了，那这个环境变量可能已经设置好了。还需要能在命令行中的任何地方执行Maven相关的命令，所以应该在PATH中加上M2_HOME/bin目录。在*nix系统中，需要加一些下面这样的东西：

```
PATH=$PATH:$M2_HOME/bin
```

在Windows系统中是这样的：

```
PATH=%PATH%;%M2_HOME%\bin
```

现在可以带着-version参数执行Maven（mvn），以确保基本安装可用。

```
mvn -version
```

应该能见到Maven输出了类似下面这种信息：

```
Apache Maven 3.0.3 (r1075438; 2011-02-28 17:31:09+0000)
Maven home: C:\apache-maven-3.0.3
Java version: 1.7.0, vendor: Oracle Corporation
Java home: C:\Java\jdk1.7.0\jre
Default locale: en_GB, platform encoding: Cp1252
OS name: "windows xp", version: "5.1", arch: "x86", family: "windows"
```

如你所见，Maven批量输出了很多实用的配置信息，这样你就知道Maven及其依赖项在你的平台上都OK了。

提示　主流IDE（Eclipse、IntelliJ和NetBeans）都支持Maven，所以熟悉了Maven在命令行中的使用方法之后，可以直接切换到IDE集成的版本。

现在Maven已经装好了，该去看看用户设置放在哪里了。为了触发用户设置目录的创建，需要确保Maven插件已经下载并安装好了。执行起来最简单的是帮助（Help）插件。

```
mvn help:system
```

这会下载、安装、并运行帮助插件，它给出的信息要比mvn -version还多。还会确保.m2目录已经创建好了。知道用户设置放哪里很重要，因为有那么几次你可能需要编辑用户设置，比如让Maven能用在一个代理服务器后面。home目录（我们会用$HOME表示）中能看到表A-1中列出的目录和文件。

表A-1　Maven用户目录和文件[1]

题　材	解　释
$HOME/.m2	包含Maven用户配置的隐藏目录
$HOME/.m2/settings.xml	包含用户特定配置的文件。在这个文件中可以指定旁路代理、私有资源库以及定制Maven行为的其他信息
$HOME/.m2/repository/	Maven的本地资源库。当Maven从Maven Central（或其他的远程Maven资源库）下载插件或依赖项时，它会在本地资源库中保存一份副本。在你用`install`目标安装本地依赖项时也是这样。这样Maven就可以用本地副本，而不用每次都去下载了

注意，用.m2目录还是因为要保持跟Maven 2的向后兼容（而不是你认为的.m3目录）。

现在已经装好了Maven，也知道用户配置在哪里了，可以开始构建java7developer了。

A.3　构建 java7developer

这一节会从几个一次性步骤开始，为构建做好准备[2]。这包括手动安装类库、重命名属性文件并编辑它，指向Java 7的本地安装。

然后就是最常见的Maven构建周期目标（`clean`、`compile`和`test`）。第一个构建期目标（`clean`）用来清理上一次构建遗留下来的工件。

Maven的构建脚本是POM（Project Object Model，项目对象模型）文件。这些POM文件就是XML文件，每个Maven项目或模块都有一个对应的pom.xml文件。POM文件即将会对备选语言提供支持，满足你所需要的更强的灵活性（很像Gradle）。

要用Maven执行构建，可以让它执行一个或几个表示特定任务（比如编译源码、运行测试等）的目标。目标全部都是绑定到默认构建周期中的，所以如果要求Maven运行一些测试（如`mvn test`），它会在试图运行测试之前把主源码和用于测试的源码都编译一下。简言之，它会迫使你遵循正确的构建周期。

让我们从一个一次性的准备任务开始吧。

A.3.1　一次性的构建准备工作

要成功运行构建，需要先重命名属性文件并编辑。如果在读12.2节时你没这么做，请转到$BOOK_CODE目录下，将sample_<os>_build.properties文件（os是你的操作系统）另存为build.properties，修改`jdk.javac.fullpath`属性，将其值指向Java 7的本地安装。这可以保证Maven构建Java代码时能选择正确的JDK。

准备工作做好了，可以运行`clean`目标了，执行构建时应该总是把它包括在内。

---

① 向Sonatype致敬，引自*Maven: the Complete Reference*在线手册（www.sonatype.com/Request/Book/Maven-The-Complete-Reference）。

② 尽管Maven构建工具最近有改进，并且也支持多语言编程，但还是有些差距。

A.3.2 clean

clean目标仅仅是把target目录删掉。要看实际效果，请切换到$BOOK_CODE目录并执行clean目标。

```
cd $BOOK_CODE
mvn clean
```

这时候，你会看到控制台中满是Maven下载各种插件和第三方类库的输出信息。Maven需要这些插件和类库运行目标，它默认从Maven Central（这些工件的主要在线资源库）下载。java7developer项目还配置了另外一个资源库，以便可以下载asm-4.0.jar文件。

注意　Maven偶尔也会为其他目标执行这个任务，所以在执行其他目标时看到它“下载互联网”不要大惊小怪。这些东西它只会下载一次。

除了“正在下载……”的信息，应该还能在控制台中看到类似下面这种信息：

```
[INFO] ----------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ----------------------------------------------------------------
[INFO] Total time: 1.703s
[INFO] Finished at: Fri Jun 24 13:51:58 BST 2011
[INFO] Final Memory: 6M/16M
[INFO] ----------------------------------------------------------------
```

如果clean目标失败了，很可能是代理服务器阻止你访问Maven Central，使你无法下载插件和第三方类库。要解决这个问题，只需修改$HOME/.m2/settings.xml文件，加上下面这些配置，为各种元素填上恰当的值。

```
<proxies>
  <proxy>
    <active>true</active>
    <protocol></protocol>
    <username></username>
    <password></password>
    <host></host>
    <port></port>
  </proxy>
</proxies>
```

重新运行这个目标，BUILD SUCCESS消息如期而至。

提示　跟其他Maven构建周期目标不同，clean不会自动调用。如果你想清除上一次构建产生的工件，必须把clean目标包括在内。

现在已经把以前构建的残留物都清除掉了，一般接下来要执行的构建周期目标是编译代码。

A.3.3 compile

compile目标用pom.xml文件中的compiler插件配置编译在src/main/java、src/main/scala和src/main/groovy目录下的源码。这实际上是将compile-scoped的依赖项加到CLASSPATH上执行Java、Scala和Groovy编译器（javac、scalac和groovyc）。Maven也会处理src/main/resources下的资源，确保它们出现在编译时的CLASSPATH中。

编译好的类会放到target/classes目录下。要看实际效果，请执行下面的目标：

```
mvn compile
```

compile目标执行起来应该很快，控制台的输出看起来应该像下面这样。

```
...
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 119 source files to
     C:\Projects\workspace3.6\code\trunk\target\classes
[INFO] [scala:compile {execution: default}]
[INFO] Checking for multiple versions of scala
[INFO] includes = [**/*.scala,**/*.java,]
[INFO] excludes = []
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\java:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\target\generated-sources\groovy-
     stubs\main:-1: info: compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\groovy:-1: info:
     compiling
[INFO] C:\Projects\workspace3.6\code\trunk\src\main\scala:-1: info: compiling
[INFO] Compiling 143 source files to
     C:\Projects\workspace3.6\code\trunk\target\classes at 1312716331031
[INFO] prepare-compile in 0 s
[INFO] compile in 12 s
[INFO] [groovy:compile {execution: default}]
[INFO] Compiled 26 Groovy classes
[INFO]-----------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ----------------------------------------------------------------
[INFO] Total time: 43 seconds
[INFO] Finished at: Sun Aug 07 12:25:44 BST 2011
[INFO] Final Memory: 33M/79M
[INFO] ----------------------------------------------------------------
```

在这一阶段，在src/test/java、src/test/scala和src/test/groovy目录下的测试类还没有编译。尽管针对它们有专门的test-compile目标，但更典型的方式是让Maven运行test目标。

A.3.4 test

运行test目标能看到Maven的构建周期的真实效果。在要求Maven测试时，它知道自己需要把之前的构建周期目标全都执行过之后才能成功运行test目标（包括compile、test-compile，还有很多其他的）。

Maven会通过Surefire插件，用pom.xml中配置为test-scoped依赖项的测试提供者（此处为

JUnit）运行测试。Maven不仅运行测试，还会生成报告文件，供以后进行分析，调研失败测试并收集测试指标。

要看实际效果，执行如下目标：

```
mvn clean test
```

Maven一旦完成测试类的编译和运行，就应该能看到类似下面这种输出的报告。

```
...
Running com.java7developer.chapter11.listing_11_3.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_4.TicketRevenueTest
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Running com.java7developer.chapter11.listing_11_5.TicketTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec

Results :

Tests run: 20, Failures: 0, Errors: 0, Skipped: 0
[INFO]----------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]----------------------------------------------------------------
[INFO] Total time: 16 seconds
[INFO] Finished at: Wed Jul 06 13:50:07 BST 2011
[INFO] Final Memory: 24M/58M
[INFO]----------------------------------------------------------------
```

测试结果保存在target/surefire-reports里。你现在可以去看看这个文本文件，能看到测试成功通过了。

A.4　小结

如果能一边看书一边运行书中的源码示例，你会对书中的内容有更深刻的认识。如果你喜欢冒险，还可以改一改我们的代码，甚至加一些新代码，然后以相同的方式编译和测试。

像Maven 3这样的构建工具，其底层实现复杂得超乎想象。如果你想深入了解这一主题，请阅读第12章，它讨论了构建和持续集成方面的内容。

附录 B

glob模式语法及示例

Java 7 NIO.2类库在循环遍历目录和其他类似任务中用glob模式执行过滤操作，参见第2章。

B.1 glob 模式语法

glob模式比正则表达式简单，其基本规则如表B-1所示。

表B-1 glob模式语法

语　法	描　述
*	匹配0或更多个字符
**	跨越目录匹配0或更多个字符
?	完全匹配单个字符
{}	限定一个子模式集合，进行匹配时各模式之间有隐含的OR关系，比如匹配模式A、B或C等
[]	匹配一组字符中的单个字符，或者如果字符间有连字符（-），则匹配其所限定范围的字符
\	转义符，在匹配*、?或\之类的特殊字符时使用

要进一步了解glob模式语法，请参见Oracle的在线Java教程（http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob）及`FileSystem`类的Java文档。

B.2 glob 模式示例

一些使用glob模式的基本例子有时被称为globbing，如表B-2所示。

表B-2 glob模式示例

语　法	描　述
`*.java`	匹配所有以.java结尾的字符串，比如Listing_2_1.java
`??`	匹配任意两个字符，比如ab或x1
`[0-9]`	匹配0到9之间的任意数字
`{groovy, scala}.*`	匹配所有以groovy.或scala.开头的字符串，比如scala.txt或groovy.pdf
`[]a-z, A-Z`	匹配一个大写或小写的英文字符
`\\`	匹配\字符
`/usr/home/**`	匹配所有以/usr/home/开头的字符串，比如/usr/home/karianna或/usr/home/karianna/docs

要查看更多glob模式匹配的例子，请参见Oracle的在线Java教程及`FileSystem`类的Java文档。

> **警告**　Java 7规范定义了自己的glob语义（而不是采用已有的标准）。有些可能会变成给程序员挖的坑，特别是在Unix上。比如说，同样是`rm *`，在Java 7中会移除以点（.）开头的文件，而在Unix的`rm/glob`中则不会移除这样的文件。[①]

① 在Unix的glob模式中，如果文件名以“.”开头，则这个字符必须显式匹配。因此`rm *`不会移除.profile，并且`tar c *`也不会归档所有文件，用`tar c.`会更好。——译者注

附录 C

安装备选JVM语言

本附录涵盖了三种JVM语言（Groovy、Scala和Clojure）以及Groovy的Web框架（Grails）的下载及安装指导，它们各自分别在第8章、第9章、第10章和第13章讨论。

C.1 Groovy

装Groovy相当简单，但如果你对设置环境变量不熟，或者刚接触某一操作系统，你应该会觉得这个指南很有帮助。

C.1.1 下载 Groovy

请先访问http://groovy.codehaus.org/Download下载最新的稳定版Groovy。我们的例子用的是Groovy 1.8.6，所以推荐你下载groovy-binary-1.8.6.zip文件。然后把下载好的压缩文件解压到选定的目录中。

> **警告** 跟很多Java/JVM相关软件的安装一样，在安装Groovy的目录名称中也不要有空格，否则可能会出现PATH和CLASSPATH错误。比如说，如果你用的是Windows操作系统，不要把Groovy装在C:\Program Files\Groovy\这样的目录中。

剩下没几步了，接下来需要设置环境变量。

C.1.2 安装 Groovy

完成下载和解压后，需要设置三个环境变量以有效运行Groovy。我们会看看基于POSIX的操作系统（Linux、Unix和Mac OS X）以及微软Windows。

1. 基于POSIX的操作系统（Linux、Unix、Mac OS X）

在一个基于POSIX的操作系统上，在哪里设置操作系统通常取决于打开终端窗口时运行的shell。表C-1中包含了各种POSIX操作系统shell中常见的用户shell配置文件的名称及位置。

表C-1　用户shell配置文件的常见位置

Shell	文件位置
bash	~/.bashrc和/或~/.profile
Korn (ksh)	~/.kshrc和/或~/.profile
sh	~/.profile
Mac OS X	~/.bashrc和/或~./.profile和/或~./bash_profile

用你喜欢的编辑器打开用户shell配置文件，加上三个环境变量：`GROOVY_HOME`、`JAVA_HOME`和`PATH`。

需要先设置环境变量`GROOVY_HOME`。加上下面这一行，用Groovy文件的真实位置（即解压文件的位置）换掉<安装目录>：

```
GROOVY_HOME=<installation directory>
```

在下面的例子中，我们将Groovy解压到了/opt/groovy-1.8.6中：

```
GROOVY_HOME=/opt/groovy-1.8.6
```

Groovy需要Java JDK才能运行。任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。你还需要确保环境变量`JAVA_HOME`已经设置好了。如果你已经装好了Java，这个可能也已经设置好了，如果还没有，可以添上下面这行：

```
JAVA_HOME=<path to where Java is installed>
```

在下面的例子中，我们将`JAVA_HOME`设置为/opt/java/java-1.7.0：

```
JAVA_HOME=/opt/java/java-1.7.0
```

最后，要能在命令行中的任何位置执行Groovy相关命令，所以得把`GROOVY_HOME/bin`加到`PATH`中：

```
PATH=$PATH:$GROOVY_HOME/bin
```

保存用户shell配置文件，在下次启动新shell时，这三个变量就会生效。现在为了确保基本安装可以正常工作，可以在命令行中执行带`-version`参数的groovy命令：

```
groovy -version
Groovy Version: 1.8.6 JVM: 1.7.0
```

在基于POSIX操作系统上安装Groovy就完成了。现在你可以回到第8章，编译并运行Groovy代码去了！

2. Windows

在Windows中，设置环境变量最好的方式是通过管理计算机的GUI。请按照下面这些步骤操作：

(1) 右键点击“我的电脑”，然后点击“属性”；

(2) 选择“高级”选项卡；

(3) 点击“环境变量”；

(4) 点击“新增”添加新的变量名称和值。

现在需要设置环境变量GROOVY_HOME。加上下面这一行，用Groovy文件的真实位置（即解压文件的位置）换掉<安装目录>：

```
GROOVY_HOME=<installation directory>
```

在下面的例子中，我们将Groovy解压到了C:\languages\groovy-1.8.6中：

```
GROOVY_HOME=C:\languages\groovy-1.8.6
```

Groovy需要Java JDK才能运行。任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。你还需要确保环境变量JAVA_HOME已经设置好了。如果你已经装好了Java，这个可能也已经设置好了，如果还没有，可以添上下面这行：

```
JAVA_HOME=<path to where Java is installed>
```

在下面的例子中，我们将JAVA_HOME设置为C:\Java\jdk-1.7.0：

```
JAVA_HOME=C:\Java\jdk-1.7.0
```

要能在命令行中的任何位置执行Groovy相关命令，所以得把GROOVY_HOME/bin加到PATH中：

```
PATH=%PATH%;%GROOVY_HOME%\bin
```

一直点击“确定”直到退出“我的电脑”的管理界面。在下次启动新命令行时，这三个变量就会生效。现在为了确保基本安装可以正常工作，可以在命令行中执行带-version参数的groovy命令：

```
groovy -version
Groovy Version: 1.8.6 JVM: 1.7.0
```

在Windows上安装Groovy就完成了。现在你可以回到第8章，编译并运行Groovy代码去了！

C.2　Scala

Scala环境可从www.scala-lang.org/downloads下载。写本书时的版本是2.9.1，但在你读到这儿时可能已经有新版本发布了。Scala确实倾向于在发布新版本时引入语言的新变化，所以如果你发现某些示例在（较新的）Scala上不能用，请认真检查语言的版本，并确保你为本书装的是2.9.1。

Windows用户应该下载.zip版本的，而基于Unix系统（包括Mac和Linux）的用户应该下载.tgz版本的。解压文件，放到指定的位置。跟Groovy一样，应该避免名称中包含空格的目录。

有几种办法可以在你的机器上设置Scala。最简单的可能就是设一个SCALA_HOME环境变量指向你安装Scala的目录。然后根据你的操作系统，按照C.1.2节的指令（安装Groovy的），把GROOVY_HOME全换成SCALA_HOME。

在完成环境的配置后，可以在命令行中键入scala，应该可以打开Scala交互式会话。如果没开，说明环境配置得不正确，应该重新试一次，确保SCALA_HOME和PATH的设置是正确的。

现在应该可以运行第9章的Scala代码清单和交互式的代码片段了。

C.3　Clojure

要下载Clojure，请访问http://clojure.org/找到包含最新稳定版的zip文件。我们在示例中用的是Clojure 1.2，所以如果你用的版本比较新，请注意它们可能会稍有差异。

解压刚下载的文件，并进入它刚创建好的目录。假定`JAVA_HOME`已经设置好了，`java`也在`PATH`上，那么现在你应该可以像第10章那样运行简单的REPL了，像这样：

```
java -cp clojure.jar clojure.main
```

Clojure跟这个附录里的前两种新语言不太一样，要用这门语言真的只需要clojure.jar文件。不用像Groovy和Scala那样设置任何环境变量。

在学习Clojure时，最简单的可能就是用REPL。当你开始考虑用Clojure做生产环境的部署时，需要用一个恰当的构建工具（比如第12章介绍的Leiningen）来管理应用的部署，以及Clojure本身的安装（从远程Maven资源库下载这个JAR文件）。

Clojure的基本安装有些局限性，但好在有几个非常好的Clojure跟IDE的集成。如果你用的是Eclipse，我们衷心向你推荐Eclipse的Counterclockwise插件，它很实用，也非常容易设置。

开发经验稍微丰富一点非常有用，因为在简单的REPL中开发大量代码可能有点容易分散人的注意力。但对于很多应用程序（特别是你正在学的）来说，基本的REPL就足够了。

C.4　Grails

装Grails相当简单，但如果你对设置环境变量不熟，或者刚接触某一操作系统，应该会觉得这个指南很有帮助。www.grails.org/installation上有完整的安装指导。

C.4.1　下载 Grails

请先访问www.grails.org下载最新稳定版Grails。我们在本书中用的版本是2.0.1。下载好后，请把压缩文件解压到选定的目录中。

警告　跟很多Java/JVM相关软件的安装一样，在安装Grails的目录名称中不要有空格，否则可能会出现`PATH`和`CLASSPATH`错误。比如说，如果你用的是Windows操作系统，不要把Grails装在C:\Program Files\Grails \这样的目录中。

接下来需要设置环境变量。

C.4.2　安装 Grails

在完成下载和解压后，需要设置三个环境变量以有效运行Grails。我们会看看在基于POSIX的操作系统（Linux、Unix和Mac OS X）以及微软Windows中如何设置环境变量。

1. 基于POSIX的操作系统（Linux、Unix、Mac OS X）

在一个基于POSIX的操作系统上，在哪里设置操作系统通常取决于打开终端窗口时运行的shell。表C-2中包含了各种POSIX操作系统shell中常见的用户shell配置文件的名称及位置。

表C-2　用户shell配置文件的常见位置

Shell	文件位置
bash	~/.bashrc和/或~/.profile
Korn (ksh)	~/.kshrc和/或~/.profile
sh	~/.profile
Mac OS X	~/.bashrc和/或~./.profile和/或~./bash_profile

用你喜欢的编辑器打开用户shell配置文件，加上三个环境变量：GRAILS_HOME、JAVA_HOME和PATH。

需要先设置环境变量GRAILS_HOME。加上下面这一行，用Grails文件的真实位置（即解压文件的位置）换掉<安装目录>

```
GRAILS_HOME=<installation directory>
```

在下面的例子中，我们将Grails解压到了/opt/grails-2.0.1中：

```
GRAILS_HOME=/opt/grails-2.0.1
```

Grails需要Java JDK才能运行。任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。还需要确保环境变量JAVA_HOME已经设置好了。如果已经装好了Java，这个可能也已经设置好了，如果还没有，可以添上下面这行：

```
JAVA_HOME=<path to where Java is installed>
```

在下面的例子中，我们将JAVA_HOME设置为/opt/java/java-1.7.0：

```
JAVA_HOME=/opt/java/java-1.7.0
```

最后，要能在命令行中的任何位置执行Grails相关命令，所以得把GRAILS_HOME/bin加到PATH中：

```
PATH=$PATH:$GRAILS_HOME/bin
```

保存用户shell配置文件，在下次启动新shell时，这三个变量就会生效。现在为了确保基本安装可以正常工作，可以在命令行中执行带-version参数的grails命令：

```
grails -version
Grails version: 2.0.1
```

在基于POSIX操作系统上安装Grails就完成了。现在可以回到第13章开始你的第一个Grails项目了！

2. Windows

在Windows中，设置环境变量最好的方式是通过管理计算机的GUI。请按照下面这些步骤

操作：

(1) 右键点击“我的电脑”，然后点击“属性”；

(2) 选择“高级”选项卡；

(3) 点击“环境变量”；

(4) 点击“新增”添加新的变量名称和值。

现在需要设置环境变量GRAILS_HOME。加上下面这一行，用Grails文件的真实位置（即解压文件的位置）换掉<安装目录>

```
GRAILS_HOME=<installation directory>
```

在下面的例子中，我们将Grails解压到了C:\languages\grails-2.0.1中：

```
GRAILS_HOME=C:\languages\grails-2.0.1
```

Grails需要Java JDK才能运行。任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。还需要确保环境变量JAVA_HOME已经设置好了。如果已经装好了Java，这个可能也已经设置好了，如果还没有，可以添上下面这行：

```
JAVA_HOME=<path to where Java is installed>
```

在下面的例子中，我们将JAVA_HOME设置为 C:\Java\jdk-1.7.0：

```
JAVA_HOME=C:\Java\jdk-1.7.0
```

要能在命令行中的任何位置执行Grails相关命令，所以得把GRAILS_HOME/bin加到PATH中：

```
PATH=%PATH%;%GRAILS_HOME%\bin
```

一直点击“确定”直到退出“我的电脑”的管理界面。在下次启动新命令行时，这三个变量就会生效。现在为了确保基本安装可以正常工作，可以在命令行中执行带-version参数的grails命令：

```
grails -version

Grails version: 2.0.1
```

在Windows上安装Grails就完成了。现在可以回到第13章开始你的第一个Grails项目了！

附录 D Jenkins的下载和安装

本附录讲解Jenkins的下载和安装，第12章要用到它。Jenkins的下载和安装很简单。如果你确实碰到了困难，它的维基页面https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins应该能提供帮助。

D.1 下载 Jenkins

可以从http://mirrors.jenkins-ci.org/上下载Jenkins。第12章的例子中用的是Jenkins 1.424。

常见的跟操作系统无关的Jenkins安装方式是用jenkins.war包。但如果你不太清楚如何运行自己的Web服务器（如Apache Tomcat或Jetty），可以根据自己的操作系统下载独立的安装包。

提示 Jenkins团队推出新版本的频率令人印象深刻，对于那些想有更稳定的CI服务器的团队来说，这可能让他们觉得内心忐忑。Jenkins团队注意到了这个问题，所以他们现在推出了一个长期支持版本（Long Term Support，LTS）。

接下来，你需要按几个简单的步骤来安装Jenkins。

D.2 安装 Jenkins

不管选的是WAR文件还是独立安装包，下载完之后需要把Jenkins装上。Jenkins要有Java JDK才能运行，任何大于1.5的版本都行（此时你很可能已经装上JDK 1.7了）。我们会先介绍WAR的安装，然后是独立安装包。

D.2.1 运行 WAR 文件

可以在命令行上运行下面的命令直接执行Jenkins WAR文件，这样安装Jenkins非常快：

```
java -jar jenkins.war
```

这种安装方法仅适用于Jenkins的快速试用，因为它很难对其他相关的Web服务器参数进行配置，所以运行起来可能不会那么顺畅。

D.2.2　安装 WAR 文件

对于更长久的安装，需要把WAR文件部署到你使用的Web服务器上（支持Java Web应用）。对于java7developer项目而言，我们只要把jenkins.war文件黏贴到Apache Tomcat 7.0.16服务器的webapps目录下就行了。

如果你对WAR文件和能支持Java Web应用的Web服务器不熟悉，可以用独立安装包。

D.2.3　安装独立安装包

独立安装包装起来也很简单。要在Windows上安装，先解压jenkins-<version>.zip文件，然后运行exe或msi安装文件。要在各种Linux上安装，需要运行合适的yum或rpm包管理器。而对于Mac OS X而言，要运行pkg文件。

无论哪种情况，都可以选择安装文件给出的默认选项，或者按自己的想法选择安装路径或其他设置。

默认情况下，Jenkins会把配置信息和任务保存在用户的主目录（我们用$USER表示）中的.jenkins目录下。要编辑UI之外的任何配置，也可以到这个目录下。

D.2.4　Jenkins 的首次运行

用你喜欢的浏览器访问Jenkins仪表板（地址通常是http://localhost:8080/或http://localhost:8080/jenkins），装的对不对一看就知道了。安装正确的话应该能见到跟图D-1类似的界面。



图D-1　Jenkins仪表板

装好了Jenkins，现在可以开始创建Jenkins任务了。请回到第12章关于Jenkins的章节去吧！

附录 E

java7developer：Maven POM

本附录涵盖了第12章中用来构建java7developer的pom.xml文件，在pom.xml文件的重要部分上展开，以便你能理解整个构建。基本项目信息（12.1节）和环境配置（代码清单12-4）在第12章的讨论已经很充分了，所以我们在这里会讨论POM的下面两个部分：

- 构建配置；
- 依赖项。

我们先从最长的那一部分开始，即构建配置。

E.1 构建配置

构建部分（`<build>`）包含一些插件及其配置信息，需要靠它们来执行Maven构建周期目标。对于大多数项目来说，这一部分通常都相当短，因为一般用默认插件的默认设置就够了。但对于java7developer项目而言，`<build>`部分包含了几个覆盖了默认设置的插件。我们之所以这样做，是为了让java7developer项目可以：

- 构建Java 7代码；
- 构建Scala和Groovy代码；
- 运行Java、Scala和Groovy测试；
- 提供Checkstyle和FindBugs代码指标报告。

如果你的构建中还有更多需要配置的地方，可以在http://maven.apache.org/plugins/index.html找到完整的插件列表。

代码清单E-1是java 7 developer项目的构建配置。

代码清单E-1　POM：构建信息

```
<build>
  <plugins>

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
```

❶ 指明要用的插件

```
    <configuration>
      <source>1.7</source>
      <target>1.7</target>
      <showDeprecation>true</showDeprecation>
      <showWarnings>true</showWarnings>
      <fork>true</fork>
      <executable>${jdk.javac.fullpath}</executable>
    </configuration>
  </plugin>

  <plugin>
    <groupId>org.scala-tools</groupId>
    <artifactId>maven-scala-plugin</artifactId>
    <version>2.14.1</version>
    <executions>
      <execution>
        <goals>
          <goal>compile</goal>
          <goal>testCompile</goal>
        </goals>
      </execution>
    </executions>
    <configuration>
      <scalaVersion>2.9.0</scalaVersion>
    </configuration>
  </plugin>

  <plugin>
    <groupId>org.codehaus.gmaven</groupId>
    <artifactId>gmaven-plugin</artifactId>
    <version>1.3</version>
    <dependencies>
      <dependency>
        <groupId>org.codehaus.gmaven.runtime</groupId>
        <artifactId>gmaven-runtime-1.7</artifactId>
        <version>1.3</version>
      </dependency>
    </dependencies>
    <executions>
      <execution>
        <configuration>
          <providerSelection>1.7</providerSelection>
        </configuration>
        <goals>
          <goal>generateStubs</goal>
          <goal>compile</goal>
          <goal>generateTestStubs</goal>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
</plugin>

<plugin>
  <groupId>org.codehaus.mojo</groupId>
```

❷ 编译Java 7代码

❸ 设置编译器选项

❹ 设置javac的路径

❺ 强制Scala编译

```
      <artifactId>properties-maven-plugin</artifactId>
      <version>1.0-alpha-2</version>
      <executions>
        <execution>
          <phase>initialize</phase>
          <goals>
            <goal>read-project-properties</goal>
          </goals>
          <configuration>
            <files>
              <file>${basedir}/build.properties</file>
            </files>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>

  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.9</version>
    <configuration>
      <excludes>
      <exclude>
      com/java7developer/chapter11/listing_11_2
➥    /TicketRevenueTest.java
        </exclude>
        <exclude>
      com/java7developer/chapter11/listing_11_7
➥    /TicketTest.java
        </exclude>
        ...
      </excludes>
    </configuration>
  </plugin>

  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>2.6</version>
    <configuration>
      <includeTestSourceDirectory>
        true
      </includeTestSourceDirectory>
    </configuration>
  </plugin>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>findbugs-maven-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
```



在测试上运行**Checkstyle**

```
        <findbugsXmlOutput>true</findbugsXmlOutput>
        <findbugsXmlWithMessages>
          true
        </findbugsXmlWithMessages>
        <xmlOutput>true</xmlOutput>
      </configuration>
    </plugin>

</build>
```


因为你要将编译Java 1.5代码的默认行为变为编译Java 1.7❷，所以需要指明正在使用（特定版本）的Compiler（编译器）插件❶。

因为已经打破了惯例，所以最好加上几个其他的编译器警告选项❸。还可以指明Java 7装在哪里❹。要想让Maven得到javac的位置，只要将与操作系统对应的sample_build.properties另存为build.properties，并修改属性jdk.javac.fullpath的值即可。

为了使用Scala插件，需要确保compile和testCompile目标运行时Scala插件能够执行❺[①]。用Surefire插件可以对测试进行配置。在这个项目的配置中，排除了几个故意失败的测试❻（你会记起来自第11章的两个TDD测试）。

我们已经讨论过构建部分了，现在让我们转入POM中的另一个关键部分，依赖管理。

E.2　依赖项管理

大多数Java项目的依赖项清单都很长，java7developer也不例外。Maven可以帮你管理这些依赖项，它在Maven Central Repository中存了数量庞大的第三方类库。重要的是，这些第三方类库都有它们自己的pom.xml文件，其中又声明了它们各自的依赖项，Maven由此可以推断出你还需要哪些类库并下载它们。

最初会用到两个主要作用域是compile和test[②]。这跟把JAR文件放到CLASSPATH中编译代码然后运行测试效果是完全一样的。

代码清单E-2为java7developer项目pom文件中的<dependencies>部分。

代码清单E-2　POM：依赖项

```
<dependencies>

    <dependency>
      <groupId>com.google.inject</groupId>
      <artifactId>guice</artifactId>
      <version>3.0</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>javax.inject</groupId>
      <artifactId>javax.inject</artifactId>
```


① 希望这个插件的后续版本能自动挂到这些目标上。

② J2EE/JEE项目中通常还会有些声明为runtime作用域的依赖项。

```
      <artifactId>javax.inject</artifactId>
      <version>1</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy-all</artifactId>
      <version>1.8.6</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>3.6.3.Final</version>
      <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.ow2.asm</groupId>
      <artifactId>asm</artifactId>
      <version>4.0</version>
      <scope>compile</scope>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>                          ❸ test作用域
    </dependency>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <version>1.8.5</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.scalatest</groupId>
      <artifactId>scalatest_2.9.0</artifactId>
      <version>1.6.1</version>
      <scope>compile</scope>                       ❹ compile作用域
    </dependency>
    <dependency>
      <groupId>org.hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>2.2.4</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javassist</groupId>
      <artifactId>javassist</artifactId>
      <version>3.12.1.GA</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

为了让Maven找到我们引用的工件，需要给出正确的<groupId>、<artifactId>和<version>❶。我们在前面说过了，将<scope>设为compile❷会将那些JAR文件加到编译代码所用的CLASSPATH中。

将<scope>设为test❸会确保Maven编译和运行测试时将这些JAR文件加到CLASSPATH中。scalatest类库是其中比较奇怪的，它应该放在test作用域中，但要放在compile作用域❹才能用。[①]

Maven pom.xml文件并不像我们所期望的那么紧凑，但我们执行的是三种语言的构建（Java、Groovy和Scala），还能生成报告。希望随着对这一领域的工具支持不断改善，Maven构建脚本能变得更精简。

① 希望这个插件的后续版本能解决这个问题。

“我自认为是一名Java专家：用Java写了15年程序，发表了几百篇文章，在各种会议中演讲，还执教Java高级课程。可阅读Ben和Martijn的这本大作，经常能给我一些意料之外的启发。”

——Heinz Kabutz博士
知名Java技术教育家、The Java Specialists’Newsletter创始人

“如果你想在Java专业领域占有一席之地，本书绝对值得拥有。”

——Stephen Harrison
FirstFuel软件公司首席软件架构师

“本书为那些对于编程有极大热情的Java开发人员提供了绝佳的资源。”

——亚马逊读者

“本书最棒的部分是依赖注入、多语言编程还有现代并发……老实说，这本书的所有内容都很棒！”

——亚马逊读者

今天，掌握JVM上的新语言对Java开发人员的意义非比寻常。因此本书除了深入探讨Java关键技术及Java 7最新特性，还用较大篇幅全面讨论了JVM上的多语言开发和项目控制，包括Groovy、Scala和Clojure这些优秀的新语言。这些技术可以帮助Java开发人员构建下一代商业软件。Java开发人员若要修炼进阶，本书绝对不容错过！



MANNING

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/Java

人民邮电出版社网址：www.ptpress.com.cn



The Well-Grounded Java Developer
Vital Techniques of Java 7 and Polyglot Programming

Java程序员修炼之道

Benjamin J. Evans 伦敦Java用户组发起人、Java社区过程执行委员会成员。他拥有多年Java开发经验，现在是一家面向金融业的Java技术公司的CEO。

Martijn Verburg jClarity的CTO、伦敦Java用户组领导人。作为一名技术专家和众多初创企业的OSS导师，他拥有十多年的经验。Martijn经常应邀出席Java界的大型会议（JavaOne、Devoxx、OSCON、FOSDEM等）并发表演讲，人送雅号“开发魔头”，赞颂他敢于向行业现状挑战的精神。

吴海星 具有10多年的Java软件开发经验，熟悉Java语言规范、基于Java的Web软件开发以及性能调优，曾获SCJP及SCWCD证书。



ISBN 978-7-115-32195-4
定价：89.00元